Министерство образования и науки РФ
ФГБОУ ВО «Томский государственный университет
систем управления и радиоэлектроники»
Кафедра безопасности информацинных систем (БИС)

А.С. Романов

СИСТЕМНОЕ ПРОГРАММИРОВАНИЕ

Методические указания по лабораторным работам, практическим занятиям, самостоятельной и индивидуальной работе

Томск – 2018

Романов А.С. Системное программирование: Методические указания по лабораторным работам, практическим занятиям и самостоятельной работе. – Томск: В-Спектр, 2018. – 129 с.

Учебное пособие содержит описания лабораторных работ и задания к практическим занятиям с примерами выполнения, требования по представлению отчётности, вопросы для самоконтроля по дисциплине «Системное программирование».

Предназначено для студентов спецальностей: 10.03.01 – «Информационная безопасность», 10.05.02 – «Информационная безопасность телекоммуникационных систем», 10.05.04 – «Информационно-аналитические системы безопасности», 10.05.03 – «Информационная безопасность автоматизированных систем», 38.05.01 – «Экономическая безопасность».

СОДЕРЖАНИЕ

Тема № 1 4
Программирование на языке Ассемблер 4
Тема № 2 21
Основные команды Ассемблера 21
Тема № 3 32
Комбинированные программы. Связывание разноязыковых модулей 32
32
Тема № 4 39
Процессы 39
Тема № 5 58
Потоки 58
Тема № 6 78
Синхронизация потоков и процессов 78
Тема № 7 109
Программирование сокетов 109
Вопросы к контрольным работам, зачету и экзамену 123
Темы индивидуальных заданий для самостоятельной работы 126
Литература 127

Тема № 1

Программирование на языке Ассемблер

Цель работы

Познакомиться со структурой программы на языке Ассемблер, разновидностями и назначением сегментов, способами организации простых и сложных типов данных, изучить форматы и правила работы с транслятором MASM или TASM, компоновщиком и отладчиком CV, познакомиться со средой программирования RadASM, возможностями Visual Studio для работы с Ассембером и средствами создания программ на Ассемблере для ОС Linux.

Краткие теоретические сведения

1.1 Ассемблеры

Ассемблеры, как правило, специфичны конкретной архитектуре, операционной системе и варианту синтаксиса языка. Вместе с тем существуют мультиплатформенные или вовсе универсальные ассемблеры, которые могут работать на разных платформах и операционных системах. Среди последних можно также выделить группу кросс-ассемблеров, способных собирать машинный код и исполняемые модули (файлы) для других архитектур и ОС.

Процесс трансляции программы на языке ассемблера в объектный код принято называть ассемблированием. В отличие от компилирования, ассемблирование - более или менее однозначный и обратимый процесс. В языке ассемблера каждой мнемонике соответствует одна машинная инструкция, в то время как в языках программирования высокого уровня за каждым выражением может скрываться большое количество различных инструкций.

1.2. Структура программы

1.2.1 Директивы сегментации

Программа на языке Ассемблер состоит из следующих частей - сегментов:

- сегмент стека, промежуточная область памяти с методом доступа к элементам LIFO («последним пришел - первым вышел»):

- сегмент данных содержит обрабатываемые, промежуточные данные.

Данные описываются следующим образом:

```
Имя Тип выражение
```

Тип определяет размер данных: DB - один байт, DW - два байта, DD - четыре байта и так далее. Выражение определяет значение, если стоит «?», то переменная не инициализирована.

Например:

```
var1 db 23            ; переменная var1 имеет значение 23
var2 dw 43h           ; var2 имеет значение 43h
var3 dd 1234567890    ; var3 имеет значение 1234567890
var4 dq ?             ; значение переменной var4 не определено -
                      ; сегмент кода содержит исполняемый код.
```

1.2.2 Использование стандартных директив сегментации

При использовании стандартных директив сегментации каждый сегмент данных необходимо описывать директивами сегментации:

```
Имя_сегмента SEGMENT Тип_выравнивания Тип_комбинирования
Класс Размер
Имя сегмента ENDS
```

Таблица 1.1 - Описание стандартных директив сегментации

Атрибут	Назначение
Выравнивание	Сообщает компоновщику о том, что нужно обеспечить размещение начала сегмента на заданной границе
Комбинирование	Сообщает компоновщику, как нужно комбинировать сегменты различных модулей, имеющие одно н то же имя
Класс	Заключенная в кавычки строка, помогающая компоновщику определить нужный порядок следования сегментов при сборке программы из сегментов нескольких модулей
Размер	Сегменты могут быть 16- или 32-разрядными

Для того что бы сообщить транслятору какой сегмент к какому сегментному регистру привязан используется директива ASSUME:

```
ASSUME регистр: Имя_сегмента, регистр: Имя_сегмента, ...
```

1.2.3. Структура программы

Структура программы с использованием директив сегментации будет выгляднть следующим образом:

1. Объявление сегмента стека.
2. Объявление сегмента данных.
3. Начало сегмента кода:

- директива **SEGMENT**;
- директива **main (proc)** (объявление главное процедуры);

- директива **ASSUME** (связывание сегментов с сегментными регистрами)

4. Настройка сегмента данных командами:

```
mov ax, Имя_сегмента_данных
mov ds,ax
```

5. Тело программы

6. Стандартное окончание:

- процедур:

```
RET;
Имя_процедуры ENDP
```

- кодового сегмента:

```
mov ax,4C90h
int 21h
Имя_сегмента ENDS
```

- программы:

```
END Имя_главной процедуры
```

1.2.4 Использование упрощенных директив сегментации

Упрощенные директивы целесообразно использовать:

1) для простых программ

2) программ, предназначенных для связывания с программными модулями, написанными на языках высокого уровня (это позволяет компоновщику эффективно связывать модули разных языков за счет стандартизации связей и управления).

Для простых программ, содержащих по одному сегменту для кода, данных н стека, описание упрощено с помощью использования упрощенных директив сегментации и директивы указания модели памяти MODEL.

MODEL [<модификатор>] <модель памяти> [др. параметры]

Таблица 1.2 - Описание некоторых упрощенных директив сегментации

Директива	Описание
.CODE [имя]	Начало или продолжение сегмента кода
.DATA	Начало или продолжение сегмента инициализированных данных
.STACK[размер]	Начало или продолжение сегмента стека модуля.
.CONST	Начало или продолжение сегмента постоянных данных
.DATA?	Начало или продолжения сегмента неинициализированных данных. Определение данных типа near.
.FARDATA [имя]	Начало или продолжение сегмента инициализированных данных типа far.

.FARDATA? [имя]	Начало или продолжение сегмента неинициализированных данных типа far.

Обязательным параметром директивы MODEL является «модель памяти». Этот параметр определяет модель сегментации памяти для программного модуля.

Возможные значения параметра «модель памяти»:

Таблица 1.3 - Описание стандартных директив сегментации

Модель	Описание
TINY	Код и данные объединены в одну группу DGROUP. Используется для программ .com
SMALL	Код занимает один сегмент, данные объединены в одну группу DGROUP.
MEDIUM	Код занимает несколько сегментов - 1 на каждый объединяемый программный модуль. Все ссылки на передачу управления типа far. Данные объединены в одной группе, все ссылки на них типа near.
COMPACT	Код в одном сегменте; ссылка на данные типа far.
LARGE	Код в нескольких сегментах. Каждый объединяемый в одну программу модуль хранится в отдельном сегменте кода.

Таблица 1.4 - Возможные значения параметра модификатор модели памяти

Параметр	Значение
use16	сегменты выбранной модели используются как 16-битные
use32	сегменты выбранной модели используются как 32-битные
dos	программа предназначена для работы в ОС MS-DOS

Для сравнения приведем два листинга с программами на ассемблере. Функционально они одинаковы и выводят на консоль сообщение: «Hello World».

Листинг 1.1. Использование стандартных директив сегментации:

```
; сегмент данных
data segment para public 'data'
message db 'Hello World.$'
data ends

; сегмент стека
stk segment stack
db 2S6 dup ('?')
stk ends

; сегмент кода
code segment para public 'code'

main proc ; начало процедуры main
assume cs:code, ds:data, ss:stk
mov ax,data ; адрес сегмента данных в регистр ax
```

```
mov ds,ax ; ax в ds
mov ah,9
mov dx,offset message
int 21h ; вывод сообщения на экран
mov ax,4c00h     ; пересылка 4c00h в регистр ax
int 21h     ; вызов прерывания с номером 21h
main endp ; конец процедуры main
code ends ; конец сегмента кода
end main ; конец программы с точкой входа main
```

Листинг 1.2. Использование стандартных директив сегментации:

```
model small ; модель памяти
.data ; сегмент данных
message db 'Hello World.$'

.stack 256h ; сегмент стека

.code ; сегмент кода

main proc ; начало процедуры main
mov ax,@data ; заносим адрес сегмента данных в регистр ax
mov ds,ax ; ax в ds
mov ah,9
mov dx,offset message
int 21h ; вывод сообщения на экран
mov ax.4c00h ; пересылка 4c00h в регистр ax
int 21h ; вызов прерывания с номером 21h
main endp ; конец процедуры main
end main ; конец программы с точкой входа main
```

1.3. Инструменты для разработки программ на Ассемблере

Рассмотрим несколько популярных программ для разработки программ на Ассемблере.

1.3.1 MASM, TASM

Macro Assembler (MASM) - ассемблер для процессоров семейства x86. Первоначально был произведён компанией Microsoft для написания программ в операционной системе MS-DOS. MASM поддерживает широкое разнообразие макросредств и структурированность программных идиом, включая конструкции высокого уровня для повторов, вызовов процедур и чередований. Позднее была добавлена возможность написания программ для Windows.

Turbo Assembler (TASM) - программный пакет компании Borland, предназначенный для разработки программ на языке ассемблера для архитектуры x86. Кроме того, TASM может работать совместно с трансляторами с языков высокого уровня фирмы Borland. TASM позволяет транслировать исходные тексты, разработанные под MASM.

Для процесса ассемблирования необходимо набрать в командной строке:

```
> masm.exe имя_файла.asm
```

или

```
> tasm.exe имя_файла.asm
```

в зависимости от выбранного ассемблера, и если все сделано правильно, то получим объектный файл (*.obj).

Каждый ассемблер имеет свой компоновщик: для MASM это link, для TASM – tlink.

Для процесса компоновки необходимо набрать в командной строке:

- для MASM:

```
> link.exe имя_файла.obj
```

- для TASM:

```
> tlink.exe имя_файла.obj
```

в результате получаем исполняемый файл.

Для процесса отладки часто применяется программа **CodeView** – отладчик, позволяющий наглядно проследить выполнение программы полностью или пошагово, наблюдая за изменением состояния флагов и регистров.

Для запуска отладчика необходимо набрать в командной строке:

```
> cv.exe имя_файла.exe
```

Для выполнения алгоритма целиком нажать - F5. Для последовательной трассировки программы - F8.

1.3.2 RadASM

В отличие от MASM и TASM, **RadASM** представляет собой полноценную среду разработки программного обеспечения для ОС Windows, изначально предназначенную для написания программ на языке ассемблера. Имеет гибкую систему файлов настроек, благодаря чему может быть использована как среда разработки программного обеспечения на высокоуровневых языках, а также документов, основанных на языках разметки.

Остановимся на процессе создания программ в RadASM более подробно.

Проект представляет собой набор файлов. В файле «Имя_проекта.rap» (расширение – «*.rap») содержится информация о настройках проекта - файлах, входящих в проект. Помимо этого файла, проекта так же содержит файл «Имя_проекта.asm» (расширение - «*.asm») - файл, содержащий основную часть программы. Так же могут входить дополнительные файлы.

Главное окно программ при запуске выглядит следующим образом (см. рисунок 1.1).

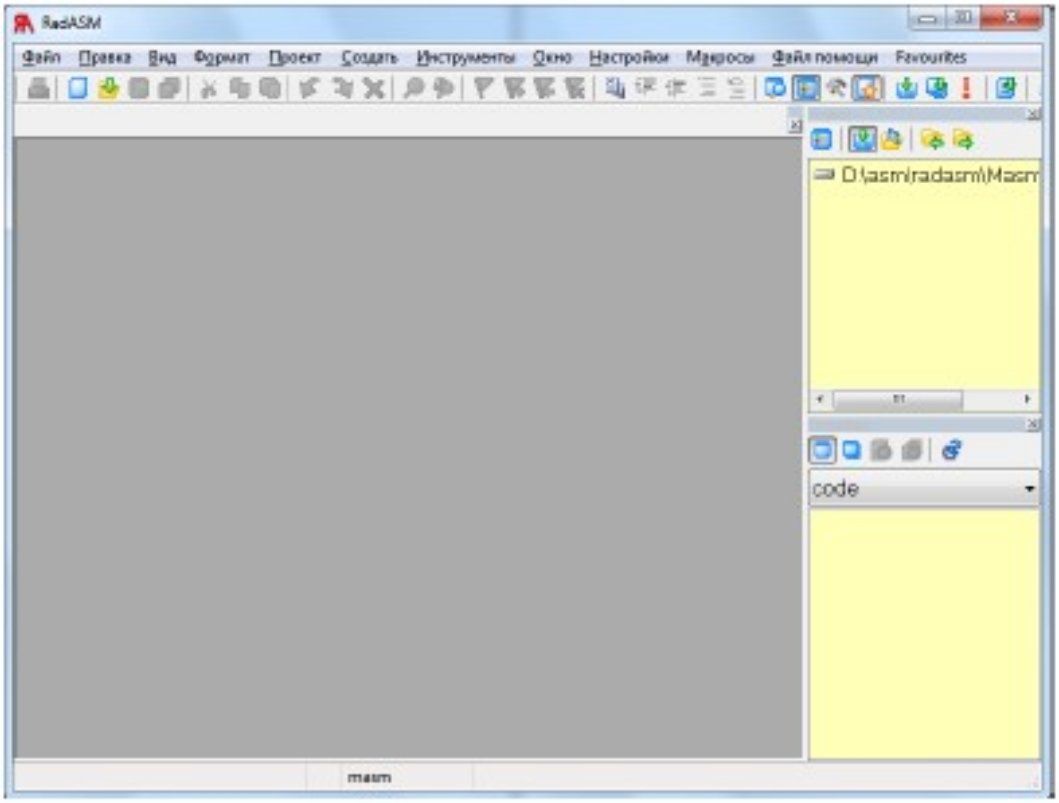

Рис. 1.1. Внешний вид среды разработки RadASM

Окно программы содержит несколько областей: панель меню, панель инструментов, рабочую область, обозреватель проектов, свойство проекта и т.д., которые можно настроить при необходимости.

Для создания нового проекта необходимо зайти в меню «Файл» и выбрать пункт «Новый проект» (см. рисунок 1.2).

В выпадающем списке «Ассемблер» необходимо выбрать строку «masm» (см. рисунок 1.3). Тип проекта - «Dos App». Ввести имя проекта. Имя проекта должно содержать одно слово без пробелов. При необходимости ввести описание. По умолчанию проекты сохраняются в папке «radasm\Masm\Projects» где radasm - имя папки, содержаий среду. При необходимости можно указать другое местоположение проекта. Нажать кнопку «Next >».

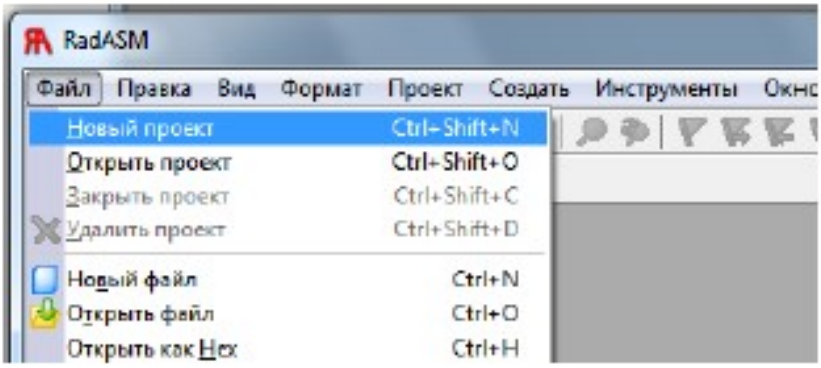

Рис. 1.2. Окно пункта меню «Файл»

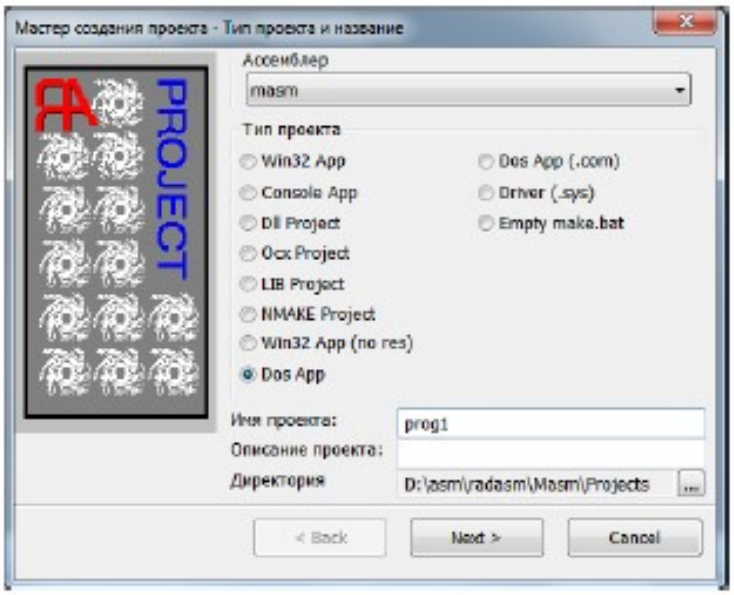

Рис. 1.3. Выбор типа проекта

Далее необходимо выбрать базовый шаблон проекта «DosExe.tpl» (см. рисунок 1.4). Шаблоны находятся в папке «radasm\Masin\Templates\». В шаблоне уже прописан базовый код, общий для всех проектов и необходимые настройки компилятора и отладчика.

После создания проекта, в редакторе будет загружена программа (см. рисунок 1.5).

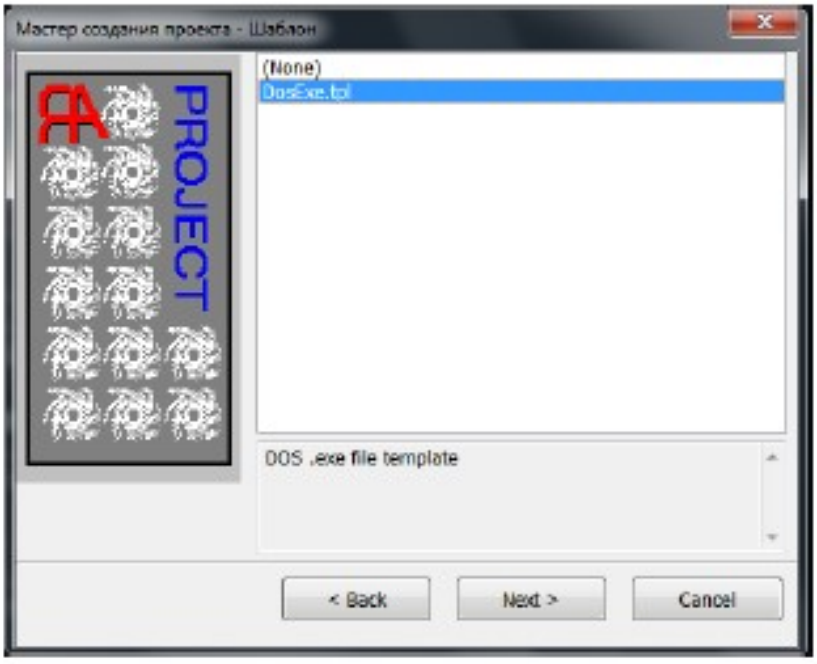

Рис. 1.4. Выбор шаблона проекта

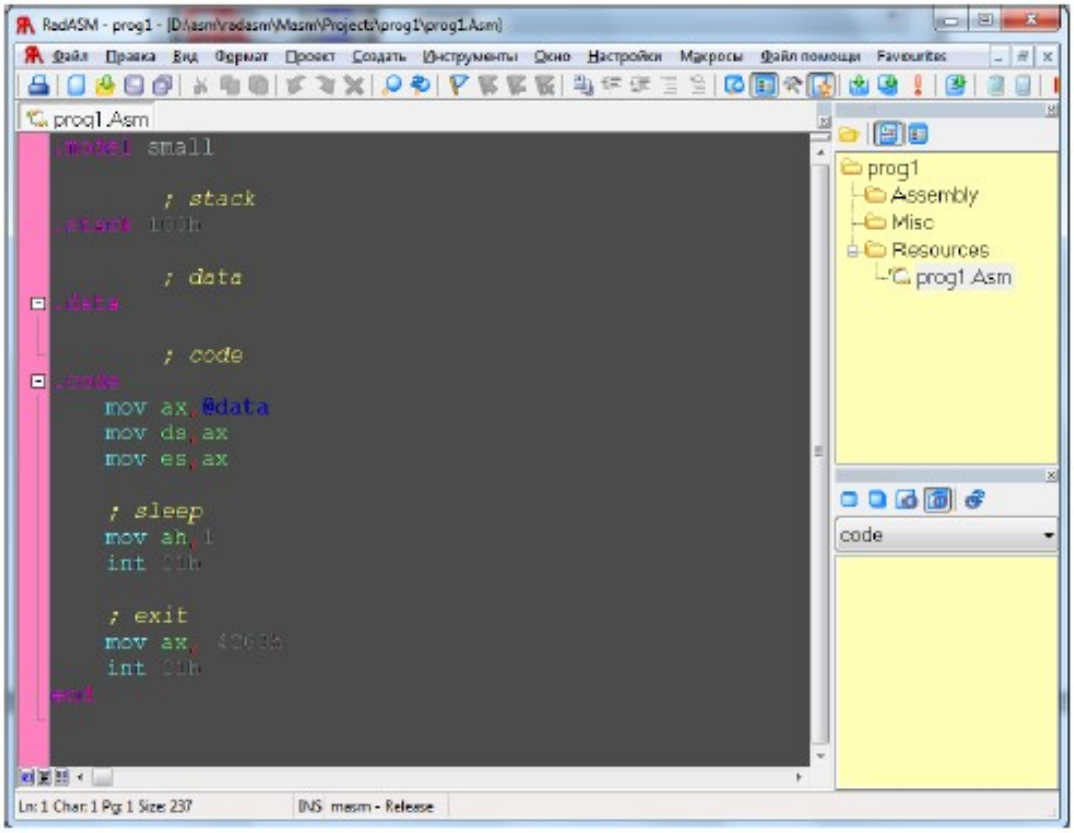

Рис. 1.5. Программа в среде разработки RadASM

После чего шаблонную программу можно отредактировать.

Листинг 1.3. Измененный текст программы:

```
.model small

.stack 256h

.data
szHello db 'Hello world','$'

.code
mov ax,@data     ;загружаем адрес сегмента данных
mov ds,ax        ;настраиваем сегмент данных
mov es,ax        ;настраиваем дополнительный сегмент

mov ah, 9        ;номер функции DOS - вывод строки на экран
lea dx,szHello   ;параметр функции - адрес строки
int 21h          ;вызов функции

mov ah,1         ;функция ожидания нажатия клавиши
int 21h          ;вызов функции
mov ax, 4C00h    ;функция завершения программы
int 21h          ;вызов функции

end
```

Откомпилируем измененную программу. Для этого в меню «Создать» выбрать пункт «Assemble» или нажать клавишу F5 (рисунок 1.6). Если ошибок не будет, то внизу в окне вывода будет подобное сообщение (рисунок 1.7):

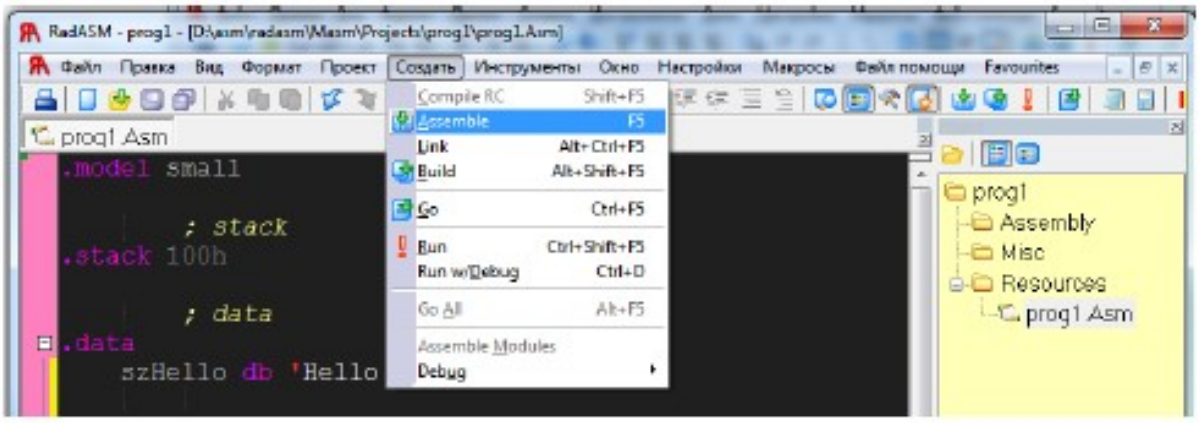

Рис. 1.6. Сборка проекта в среде разработки RadASM

Рис. 1.7. Окно вывода с информацией о ассемблировании

После ассемблирование получится файл «Имя_проекта.obj», который необходимо слинковать. Для этого в том же меню «Создать» нужно выбрать пункт «Link» или нажать сочетание клавиш Alt+Ctrl+F5 (рисунок 1.8). Если в процессе линкования не случилось ошибок, то в результате в папке проекта появится файл «Имя_проекта.exe».

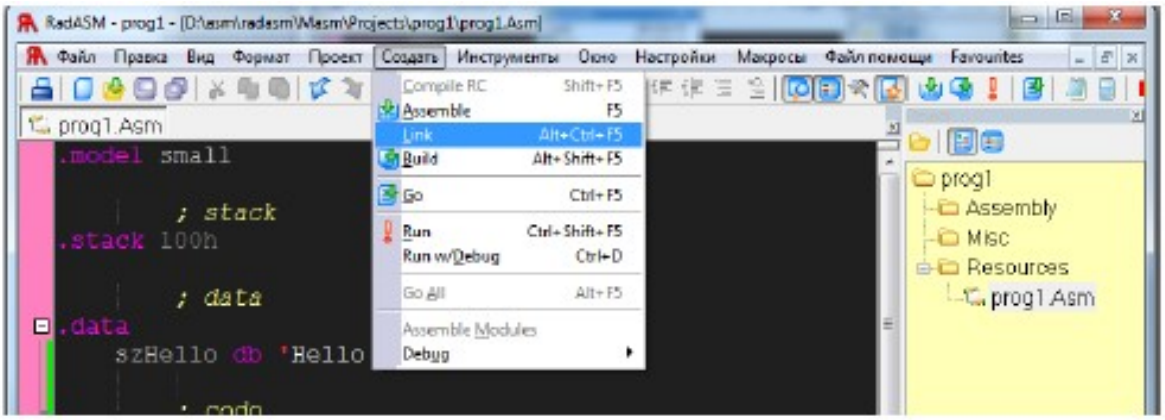

Рис. 1.8. Линковка проекта в среде RadASM

Далее запустим этот файл под отладчиком. Для этого в том же меню «Создать» выбрать пункт «Run w/Debug» или нажать сочетание клавиш Ctrl+D. Запустится отладчик с загруженной программой (рисунок 1.9). Окно отладчика состоит из нескольких частей: панель меню: окно, отображающее код программы (сегмент кода); окно сегмента данных: окно сегмента стека; окно, отображающее состояние регистров процессора и флагового регистра. В окне сегмента кода отображается адрес инструкции, машинная команда в 16-ном виде и ассемблерная мнемоника. Для того, что бы программа начала выполняться по шагам (трассировка), необходимо нажать клавишу F7 или F8. Отличие F7 от F8 в том, что в первом случаи программа будет выполнять по шагам с заходом в процедуры и пошаговом выполнении цикла. Нажмем несколько раз клавишу F7 до тех пор, пока курсор не встанет напротив инструкции **mov es.ax**. Для того, что бы увидеть сегмент данных, поставим курсор в окно сегмента данных, нажмем правой кнопкой мыши, выберем пункт «Goto ...» (или нажать сочетание клавиш Ctrl+G) (рисунок 1.10) и в открывшемся окошке введем **ds:0**, нажмем «OK» или «Enter» (рисунок 1.11). В сегменте данных видим строчку «Hello World» (рисунок 1.12).

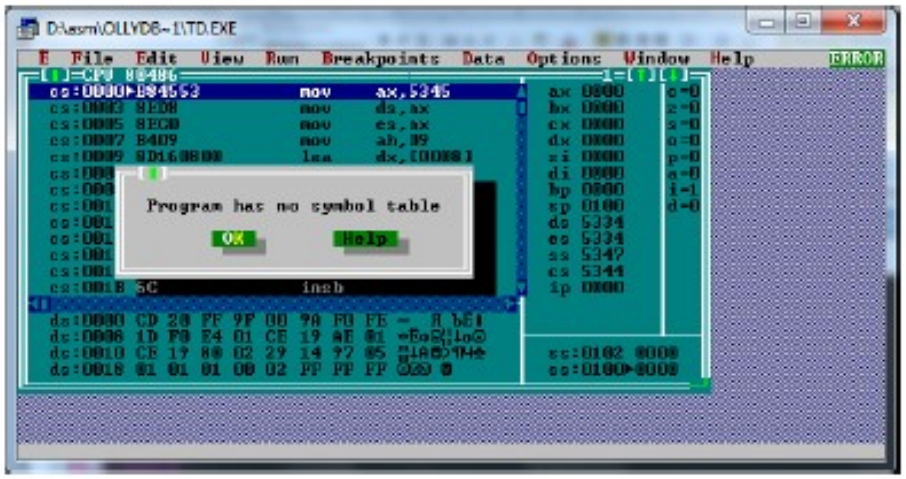

Рис. 1.9. Запущенная под отладчиком программа

Продолжим трассировку до тех пор, пока не появится предупреждение о завершении программы. В процессе отладки изменяется содержимое регистров.

Рис. 1.10. Настройка отображения сегмента данных

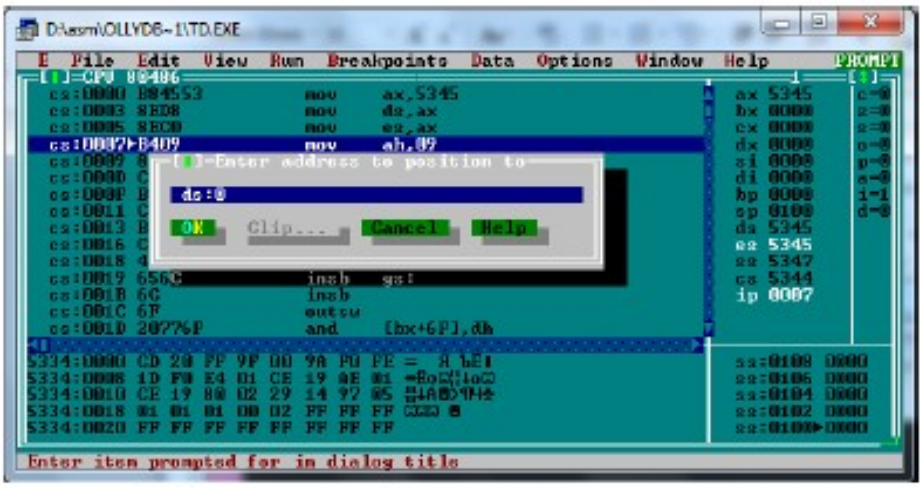

Рис. 1.11. Настройка отображения содержимого сегмента

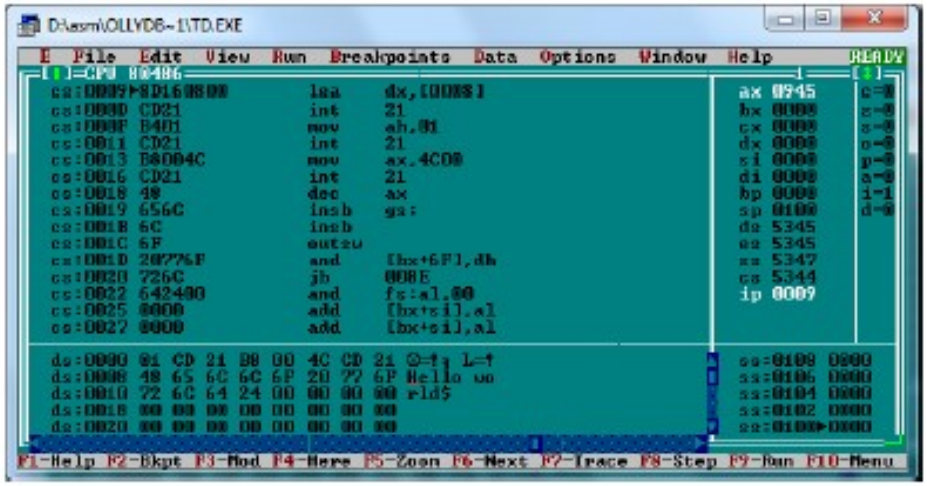

Рис. 1.12. Программа под отладчиком TurboDebugger

1.3.3 Visual Studio и Ассемблер

Создаем проект. В Visual Studio, выбираем File > New > Project. В Visual Studio нет языка ассемблер в окне выбора типа проекта, поэтому создаем C++ Win32 проект. В окне настроек нового проекта выбираем «Empty Project».

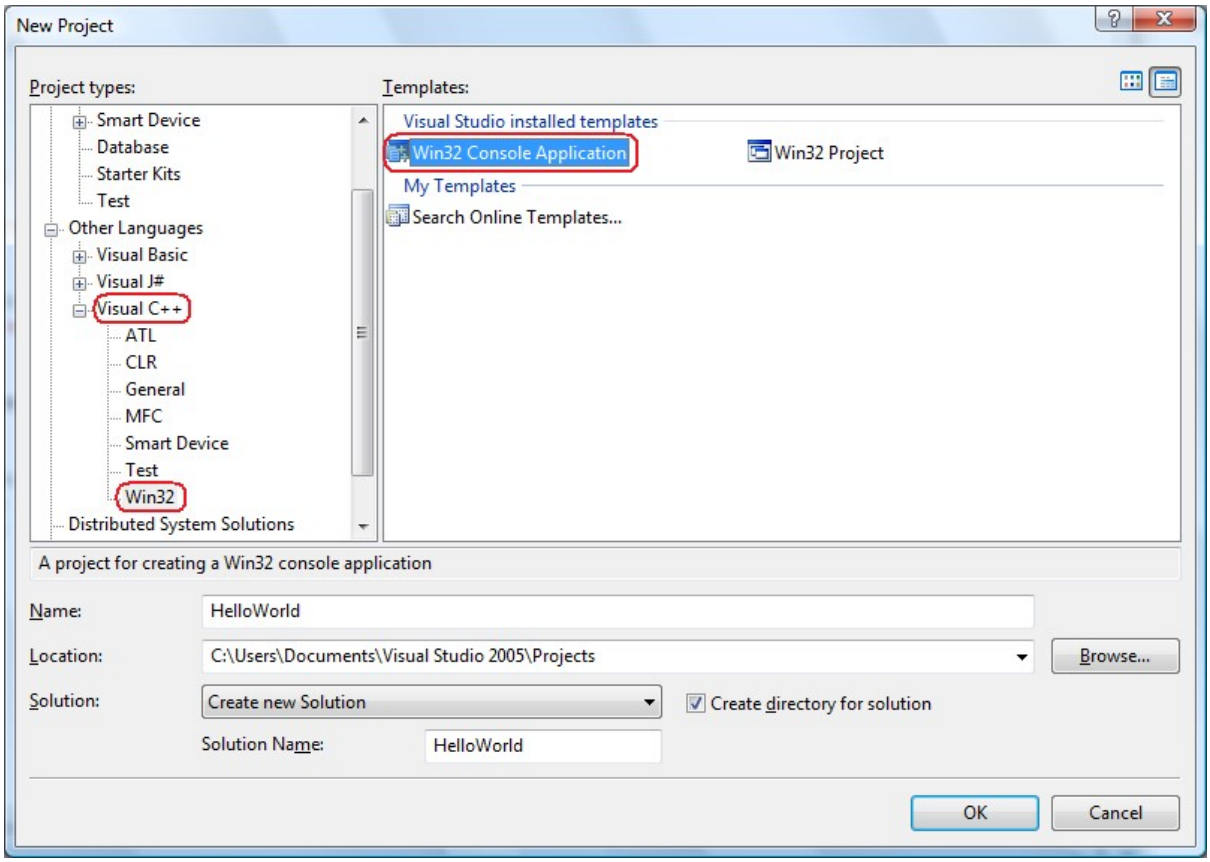

Рис. 1.13. Создание проекта

Чтобы включить поддержку Ассемблера необходимо настроить в проекте опции сборки, указав какой программой необходимо компилировать файлы *.asm, содержащие ассемблерный код. Для этого выбираем пункт меню «Custom Build Rules...».

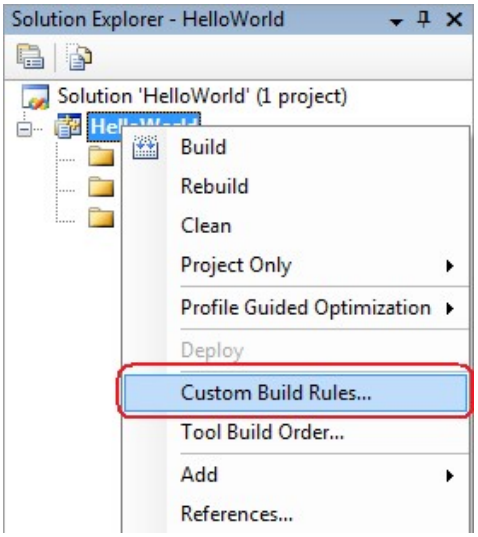

Рис. 1.14. Опции компиляции

В открывшемся окне указываем опции компиляции для различных файлов, Visual Studio уже имеет готовое правило для файлов *.asm, нам необходимо лишь включить его, установив напротив правила «Microsoft Macro Assembler» галочку.

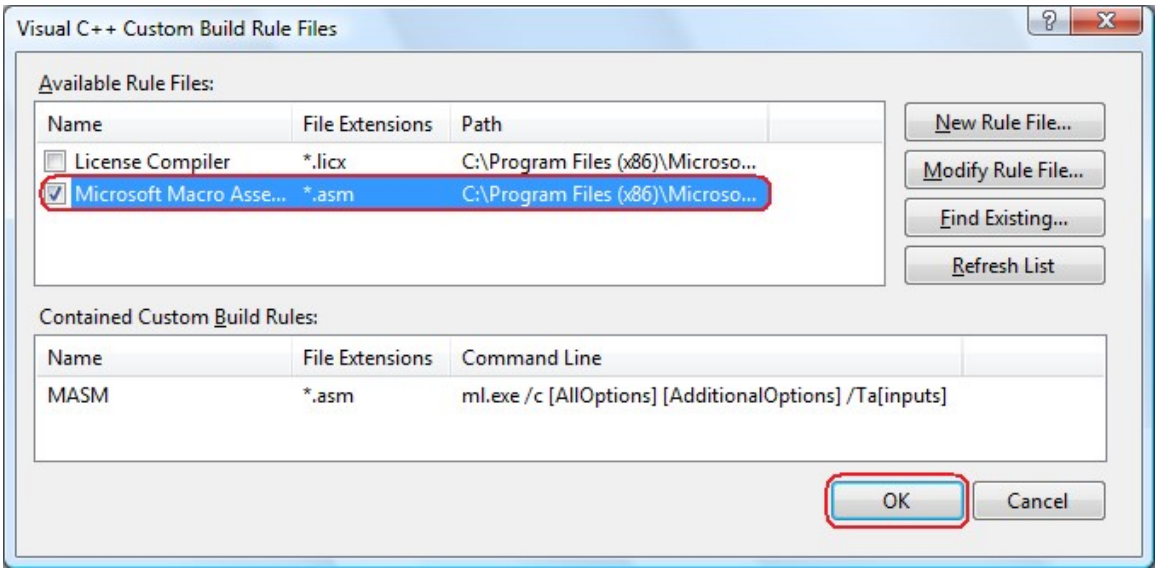

Рис. 1.15. Опции компиляции

Добавляем код. Для этого добавим в Source Folder новый файл со следующим кодом.

Листинг 1.4. Простая программа под ОС Windows:

```
.386
.mode flat, stdcall
option casemap:none
include windows.inc
include user32.inc
include kernel32.inc
includelib user32.lib
includelib kernel32.lib

.data
 hello BYTE "Hello world!",10,13,0

.code
 start:
  invoke printf, ADDR Hello, eax
  ret
end start
```

В данном случае для печати сообщения используется вызов функции printf.

Теперь запускаем проект, для этого выбираем Debug > Start Without Debugging или нажимаем комбинацию Ctrl-F5, что эквивалентно использованию команд компиляции и линковки:

```
> ML /c /coff "hello.asm"
> LINK /SUBSYSTEM:CONCOLE "hello.obj"
```

Для отладки программы можно воспользоваться стандартным отладчиком Visual Studio.

1.3.4 GCC и Ассемблер для Linux

Не стоит также забывать и о программировании для операционных систем, отличных от DOS и Windows. В листинге 1.4 рассмотрен пример программы, выводящий сообщение «Hello world», написанный для ОС Linux. Особо уделите внимания отличиям синтаксиса команд от приведенных ранее.

По умолчанию, GNU Assembler (GAS) использует AT&T-синтаксис для x86 и x86-64, то есть регистры обозначаются префиксом % и регистр-приёмник указывается после источника.

Листинг 1.4. Пример простой программы на Ассемблере для Linux:

```
.data /* описываем сегмент кода */
hello_str:
 .string "Hello, world!\n" /* строка для вывода */
 .set hello_str_length, . - hello_str - 1  /* длина строки */

.text /* описываем сегмент кода */
.globl  main     /* main - глобальный символ */
.type   main, @function    /* main - функция */

main:
 movl $4, %eax   /* поместить номер системного вызова write=4
                    в регистр %eax */
 movl $1, %ebx   /* 1ый параметр - в регистр %ebx; номер
                    файлового дескриптора stdout - 1 */
 movl $hello_str, %ecx     /* 2ой параметр - в регистр %ecx;
                              указатель на строку */
 movl $hello_str_length, %edx   /* 3ий параметр - в регистр
                                   %edx; длина строки */

 int $0x80         /* вызов прерывания 0x80, аналога 21h для DOS*/
 movl $1, %eax     /* номер системного вызова exit - 1 */
 movl $0, %ebx     /* передать 0 как значение параметра */
 int $0x80         /* вызвать exit(0) */

 .size main, . - main    /* размер функции main */
```

Сохраним текст программы в файле с расширением «*.s», например как «hello.s» и скомпилируем её с помощью компилятора GCC (сокр. от GNU Compiler Collection):

```
[user@host:~]$ gcc hello.s -o hello -g
```

Если компиляция проходит успешно, GCC ничего не выводит на экран. Кроме компиляции, GCC автоматически выполняет и компоновку. На выходе получаем исполняемый файл hello. Запустим программу и убедимся, что она корректно завершилась с кодом 0:

```
[user@host:~]$ ./hello
Hello, world!
[user@host:~]$ echo $?
0
```

Для отладки программ под ОС Linux особой популярностью пользуется отладчик GDB (сокр. от GNU Debugger):

```
[user@host:~]$ gdb hello
[user@host:~]$ info registers
```

Первая команда произведет отладку программы hello. Вторая – покажет содержимое всех регистров. Также GDB может применяться для дизассемблирования программ (команды disass и disassemble), обратной трассировки (команда bt), установки точек останова (команда break) и др.

Задание

1. Изучить теоретические сведения, самостоятельно изучить способы организации и синтаксис для сложных типов данных в Ассемблере: структур, массивов, записей.
2. Реализовать приведенные примеры программ на Ассемблере при помощи MASM или TASM, в среде программирования RadASM или любой другой среде, позволяющей программировать на Ассемблере для операционных систем DOS/Windows.
3. Подготовить образ операционной системы Linux. Установить компилятор GCC и другие необходимые пакеты.
4. Отладить простейшую программу с помощью компилятора GCC.
5. Воспользоваться отладчиком и научиться пользоваться представляемой им информацией.
6. Получить индивидуальное задание у преподавателя и реализовать соответствующую программу на Ассемблере и на языке высокого уровня.
7. Дизассемблировать обе программы, провести сравнительный анализ скорости работы программ, объема полученного дизассемблированного кода. Сделать выводы.
8. Написать отчет и защитить у преподавателя.

Варианты индивидуальных заданий

Варианты заданий отсутствуют. Учитывается привлечение дополнительной информации по использованию различных сред программирования и отладчиков и приобретенные навыки работы с ними под разными операционными системами.

Контрольные вопросы

1. Системные, прикладные, промежуточные программы.
2. Этапы подготовки программы к выполнению.

3. Исходный модуль. Объектный модуль. Программный модуль.

4. Трансляторы и трансляция. Ассемблирование, компиляция, интерпретация.

5. Понятие «программа» и «программное обеспечение». Сходства и различия.

6. Свойства программного обеспечения.

7. Понятие «операционная система» и «операционная среда». Сходства и различия.

8. Этапы разработки программного обеспечения.

9. Редактор связей.

10. Загрузчик. Функции загрузчика.

11. Языки программирования низкого и высокого уровня. Место языка Ассемблер в этой иерархии.

12. Чем отличается Intel-синтаксис ассемблера от AT&T-синтаксиса?

13. Регистры общего назначения.

14. Регистры стека.

15. Индексные регистры.

16. Регистр командного указателя.

17. Сегментные регистры.

18. Флаговый регистр.

19. Инструменты разработки программ на Ассемблере.

20. Отладчик.

Тема № 2

Основные команды Ассемблера

Цель работы

Познакомиться с основными командами языка Ассемблер, научиться применять полученные знания при решении практических задач.

Краткие теоретические сведения

2.1. Формат команд

Формат кодирования команд Ассемблера имеет следующий вид:

```
[Метка:] Команда [Операнды] [; Комментарии]
```

Метка, команда и операнды разделяются, по крайней мере, одним пробелом. Метка может содержать буквы, цифры, специальные символы начинаться с буквы. Ассемблер не делает различия между заглавными и строчными буквами.

Операнд определяет начальное значение данных или элементы, над которыми выполняется действие.

```
Метка      Команда    Операнд
const db   mov        ax,0
```

Комментарий начинается с символа точки с запятой. Все символы после точки с запятой игнорируются.

2.2. Классификация команд

В Ассемблере выделяют следующие группы команды:

1) команды обмена данными: MOV, LEA, LDS, LES;

2) арифметические команды: INC, ADD, ADC, DEC, SUB, SBB, MUL, IMUL, DIV, IDIV;

3) логические команды: AND, OR, XOR, NOT, TEST;

4) команды сдвига и вращения: SHL, SHR, SAL, SAR, ROL, ROR, RCL, RCR,;

5) команды работы с флагами: CLS, STS, CMC, CLD, STD, CLI, STI, LAHV, SAHV, PUSHF, POPF;

6) команды безусловной передачи управления: CALL, RET, JMP;

7) команды сравнения, условного перехода и циклов: CMP, JE, JNE, JL/JNGE, JLE/JNG, JG/JNLE, JGE/JNL, JB/JNAE, JBE/JNA, JA/JNBE, JAE/JNB, JC, JP, JZ, JS, JO, LOOP, LOOPE/LOOPZ, LOOPNE/LOOPNZ;

8) команды цепочек: MOVS, CPMS, SCAS, LODS, STOS, INS, OUTS;

9) команды работы со стеком: PUSH, POP, PUSHF, POPF;

10) команды работы с битами: BSF/BSR, BT, BTR, BTS, BTC;

11) команды управления состоянием микропроцессора: NOP, HLT, CLI, STI;

12) команды работы с портами ввода-вывода: IN, OUT;

13) системные вызовы: INT, INTO, IRET и др.

2.3. Примеры использования команд

Рассмотрим примеры применения некоторых из приведенных выше команд. Для начала освоим некоторые базовые способы использования самой распространенной команды MOV.

Листинг 2.1. Пример использования команды MOV:

```
.model small
.stack 100h
.data
 vara db 3
 varb db ?
 varc db ?
 varf dw 1234
 vare dd 87654321

.code
 mov  ax,@data
 mov  ds,ax
 mov  es,ax
 mov  ah,vara    ; В регистре ah - значение переменной vara
 mov  varb,ah    ; В переменной varb - значение из регистра al
 mov  bx,offset varf  ; В bx - АДРЕС ячейки памяти переменной varf
 mov  ax,[bx]    ; В ax - СОДЕРЖИМОЕ ячейки памяти, на которую
                 ; указывает регистр bx
 mov  ah,[bx]
 mov  bx,offset vare  ; bx указывает на ячейку памяти ее
 mov  a1,byte ptr [bx]
 mov  ah,byte ptr [bx+1]
 mov  dx,[bx]
 mov  varc,al
 inc  varc       ; Значение в ячейки varc увеличивается на единицу

 mov  ah,1
 int  21h
 mov  ax, 4C00h
 int  21h
```

Хорошим примером использования команды загрузки адреса LEA является пример, в котором с её помощью происходит копирование строки.

Листинг 2.2. Пример использования команды LEA:

```
.model small
.stack 100h
.data
 szStrSource db 'Строка для копирования','$'
 szStrDest db sizeof szStrSource dup (?),'$'
```

```
 szStrDest db 16 dup (?),'$'
 AddrszStrSource dd szStrSource

.code
 mov  ax,@data
 mov  ds,ax
 mov  es,ax
 lea  si,szStrSource ; si указывает на адрес первой ячейки строки
 lea  di,szStrDest ; di указывает на адрес первой ячейки строки-
приемника
 les  bx,AddrszStrSource ; полный указатель на szStrSource
 mov  cx,sizeof szStrSource ; cx число символов в строке приемнике

ml:
 mov  al,[si] ; копируем очередной символ в регистр al
 mov  [di],al ; копируем символ из регистра в строку-приемник
 inc  si   ; увеличиваем счетчики-индексы, указывающие
 inc  di   ; на очередной символ и место назначения
 loop ml

 mov  ah,9
 les  dx,AddrszStrSource
 int  21h

 mov  ah,1
 int  21h
 mov  ax, 4C00h
 int  21h
```

При использовании арифметических команд, нельзя забывать про возможное переполнение, переносы и заемы, учитывать знаки чисел и проч. Рассмотрим примеры правильного использования команд сложения INC и ADD (см. листинг 2.3), вычитания DEC и SUB (см. листинг 2.4), умножения MUL (см. листинг 2.5), деления DIV (см. листинг 2.6).

Листинг 2.3. Пример использования команд ADD и INC:

```
.model small
.stack 100h
.data
  a db 254
.code
  mov ax,@data
  mov ds,ax
  mov es,ax
  xor ax,ax ;Обнуление регистра ax
  inc ax    ;Увеличение на единицу ax
  xor ax,ax
  mov ax,142     ;142 в десятичном виде
  add ax,45h     ;45h в шестнадцатеричном виде
```

```
  xor ax,ax
  add al,17
  add al,a
  jnc m1    ;если нет переноса, то перейти
  adc ah,0  ;в ах сумма с учетом переноса
m1:
  mov ah,1
  int 21h
  mov ax, 4C00h
  int 21h
end
```

Листинг 2.4. Пример использования команд SUB и DEC:

```
.model small
.stack 100h
.data
.code
  mov ax,@data
  mov ds,ax
  mov es,ax
  xor ax,ax ; Обнуляем регистр ax
  mov al,15 ; al = 15
  dec al ; al = 14
  sub al,20 ; должно быть -6
  jnc m1 ; нет переноса?
  neg al ; в al модуль результата

m1:
  mov ah,1
  int 21h
  mov ax, 4C00h
  int 21h
end
```

Листинг 2.5. Пример использования команды MUL:

```
.model small
.stack 100h
.data
  res label word ; строка 6
  res_l db 45
  res_h db 0

.code
  mov ax,@data
  mov ds,ax
  mov es,ax
  xor ax,ax
  mov al,25
  mul res_l
```

```
  jnc m1    ; если нет переполнения, то на m1
  mov res_h,ah ; старшую часть результата в res_h

m1:
  mov res_l,al
  mov ah,1
  int 21h
  mov ax, 4C00h
  int 21h
end
```

Листинг 2.6. Пример использования команды DIV:

```
.model small
.stack 100h
.data
  del dw 298
delt db 45

.code
  mov ax,@data
  mov ds,ax
  mov es,ax
  xor ax,ax
  mov ax, del
  div delt  ; в al частное, в ah – остаток

  mov ah,1
  int 21h
  mov ax, 4C00h
  int 21h

end
```

Следующая программа (см. листинг 2.7) демонстрирует возможное применение команды логического сдвига вправо SHR и команды вращения вправо ROR в сочетании с командами других групп. Программа, преобразует число в шестнадцатеричном виде в строку - сдвигая вправо операнд, осуществляется операция деления на 2, 4, 8…

Листинг 2.7. Пример использования команд SHR и ROR:

```
.model small
.stack 100h
.data
  szad db ‘4374d =’
  a dw 4374                ; = 1116h = 0001 0001 0001 0110b
  aend db ‘ = $’
  szah db ‘000h’

.code
```

```
  mov ax, @data
  mov ds,ax
  mov es,ax

.386
  mov ah,9
  lea dx, szad
  int 21h

  lea si,a
  lea di,szah
  xor ax,ax
  mov al,[si+1]            ; al = 0001 0110b = 16h
  ror ax,4                 ; ax = 0110 0000 0000 0001b = 3001h
  shr ah,4                 ; ax = 0000 0110 0000 0001b = 601h
  or ax,0011000000110000b  ; ax = 0011 0110 0011 0001b = 3631h
  mov [di],ax
  xor ax,ax
  mov al,[si]              ; al = 11h = 0001 0001b
  ror ax,4                 ; ax = 0001 0000 0000 0001b = 1001h
  shr ah,4                 ; ax = 0000 0001 0000 0001b = 0101h
  or ax,0011000000110000b  ; ax = 0011 0001 0011 0001b = 3131h
  mov [di+2],ax

  mov ah,9
  lea dx,szah
  int 21h

  mov ah,1
  int 21h
  mov ax, 4c00h
  int 21h
end
```

Под цепочкой в Ассемблере подразумевается непрерывная последовательность байт, слов или двойных слов, обрабатываемая как единое целое. К цепочке возможен только последовательный доступ от начала к концу или от конца к началу. В качестве примера рассмотрим программу пересылки символов из одной строки в другую. Строки находятся в одном сегменте памяти. Для пересылки используется команда-примитив MOVS с префиксом повторения rep (см. листинг 2.8).

Листинг 2.8. Пример использования команды MOVS:

```
.model small
.stack 256
.data
 source db 'тестируемая строка','$'   ;строка-источник
 dest db 19 dup (' ') ;строка-приёмник
```

```
.code
 assume ds:@data,es:@data
main:
  mov ax,@data   ;загрузка сегментных регистров
  mov ds,ax      ;настройка регистров ds и es
                 ;на адрес сегмента данных
  mov es,ax
  cld ;сброс флага df — обработка строки от начала к концу
  lea si,source  ;загрузка в si смещения строки-источника
  lea di,dest    ;загрузка в ds смещения строки-приёмника
  mov cx,20      ;для префикса rep — счетчик повторений (длина)
rep movs dest,source       ;пересылка строки
  lea dx,dest
  mov ah,09h               ;вывод на экран строки-приёмника
  int 21h
exit:
  mov ax,4c00h
  int 21h
end
```

Следующая программа (см. листинг 2.9) выводит на консоль состояния флагов SF, ZF, AF, PF и CF. Вывод значений флагов оформлен в виде подпрограммы, для вызова используется команда CALL, для выхода – RET.

Листинг 2.9. Пример использования команд работы с флагами и команд безусловной передачи управления:

```
use16
org 100h
jmp start           ;Переход на метку start

; Данные
s_sf  db 'FLAGS: SF=$'
s_zf  db ' ZF=$'
s_af  db ' AF=$'
s_pf  db ' PF=$'
s_cf  db ' CF=$'
s_endldb 13,10,'$'
s_pak db 'Press any key...$'

start:
  mov al,120
  add al,56          ;Пример арифметической операции
  call print_flags   ;Вызов процедуры вывода состояния флагов
  cmc                ;Инверсия флага CF
  call print_flags   ;Вызов процедуры вывода состояния флагов
  xor ax,ax          ;Пример логической операции
  call print_flags   ;Вызов процедуры вывода состояния флагов
```

```
  mov ah,9
  mov dx,s_pak
  int 21h             ;Вывод строки 'Press any key...'
  mov ah,8            ;Функция DOS 08h - ввод символа без эха
  int 21h

  mov ax,4C00h
  int 21h             ; Завершение программы

print_flags:          ;Процедура вывода состояния флагов на консоль

  push ax
  push cx
  push dx
  pushf               ;Сохранение регистра флагов
  lahf                ;Загрузка младшего байта FLAGS в AH
  mov cl,ah           ;CL = AH
  mov ah,9            ;Функция DOS 09h - вывод строки
  mov dx,s_sf         ;DX = адрес строки 'FLAGS: SF='
  int 21h             ;Обращение к функции DOS
  shl cl,1            ;Сдвиг CL влево на 1 бит
  call print_cf       ;Печать выдвинутого бита
  mov dx,s_zf
  int 21h             ;Вывод строки ' ZF='
  shl cl,1            ;Сдвиг CL влево на 1 бит
  call print_cf       ;Печать выдвинутого бита
  mov dx,s_af
  int 21h             ;Вывод строки ' AF='
  shl cl,2            ;Сдвиг CL влево на 2 бита
  call print_cf       ;Печать выдвинутого бита
  mov dx,s_pf
  int 21h             ;Вывод строки ' PF='
  shl cl,2            ;Сдвиг CL влево на 2 бита
  call print_cf       ;Печать выдвинутого бита
  mov dx,s_cf
  int 21h             ;Вывод строки ' CF='
  shl cl,2            ;Сдвиг CL влево на 2 бита
  call print_cf       ;Печать выдвинутого бита
  mov dx,s_endl
  int 21h             ;Вывод конца строки
  popf                ;Восстановление регистра флагов
  pop dx
  pop cx
  pop ax
  ret

print_cf:  ;Процедура вывода значения флага CF в виде символа
  push ax
  push dx
```

```
  mov ah,2              ;Функция DOS 02h - вывод символа
  mov dl,'0'            ;DL = '0'
  adc dl,0              ;Если CF = 1, то в DL будет символ '1'
  int 21h               ;Обращение к функции DOS
  pop dx
  pop ax
  ret
```

Команда сравнения CMP и условного перехода JNE используется в листинге 2.10 для проверки введены ли с клавиатуры два одинаковых символа. Если символы одинаковые, программа выдаст сообщение «same», а иначе - «different».

Листинг 2.10. Пример использования команд CMP и JNE:

```
data segment;
  mes_e db 10, 13, 'Same$'
  mes_ne db 10, 13, 'Different$'
data ends

code segment
start:
  assume cs:code, ds: data
  mov ax, data
  mov ds, ax

  mov ah, 01            ;вводим первый символ и запоминаем его в bl
  int 21h
  mov bl, al

  int 21h               ;вводим второй символ

  mov ah, 09
  lea dx, mes_ne        ;пусть символы неравны
  cmp bl, al            ;сравниваем их
  jne m_ne        ;если они неравны, переходим на вывод сообщения
  lea dx, mes_e   ;иначе загружаем адрес другой строки
m_ne:
  int 21h         ;и выводим
  mov ax, 4c00h
  int 21h
code ends
end start
```

В приведенных примерах используются и команды из других групп.

Задание

1. Изучить материал лекций по теме практического занятия и приведенные выше примеры программ. Самостоятельно изучить способы орга-

низации и синтаксис для сложных типов данных в Ассемблере: структур, массивов, записей.

2. Реализовать примеры программ на Ассемблере при помощи одного из инструментов, изученных в ходе занятий по Теме №1 «Знакомство с Ассемблером».

3. Воспользоваться отладчиком и посмотреть, какие регистры, флаги изменяются при выполнении тех или иных команд.

4. Получить индивидуальное задание у преподавателя и реализовать соответствующую программу на Ассемблере. При этом исходные данные должны считываться/сохраняться из/в файла.

5. Написать отчет и защитить у преподавателя.

Варианты заданий

1. Алгоритм нахождения расстояния Левенштейна между строками.
2. Алгоритм нахождения расстояния Хэмминга между строками.
3. Алгоритм "Решето Эратосфена".
4. Кодирование и Проверка кода Хэмминга.
5. Алгоритм Евклида нахождения НОД.
6. Вычисления значения выражения, записанного в обратной польской нотации.
7. Алгоритм сортировки массива с помощью двоичного дерева.
8. Алгоритм сортировки массива Timsort.
9. Алгоритм интерполяционного поиска.
10. Алгоритм поиска с помощью золотого сечения.
11. Алгоритм обхода графа (на выбор).
12. Алгоритм Кнута-Морриса-Пратта поиска подстроки в строке.
13. Подсчет частот встречаемости слов в тексте.
14. Подсчет частот встречаемости биграмм (пар) символов в тексте.
15. Подсчет частот встречаемости триграмм (троек) символов в тексте.
16. Подсчет частот встречаемости биграмм (пар) слов в тексте.
17. Подсчет частот встречаемости триграмм (троек) слов в тексте.
18. Свой вариант.

Контрольные вопросы

1. Формат кодирования команд Ассемблера. Метка. Команда Операнды. Комментарии.
2. Что может выступать в роли операнда команды?
3. Стандартные директивы сегментации.
4. Упрощенные директивы сегментации.
5. Методы адресации.
6. Прямая адресация.
7. Непосредственная адресация.

8. Косвенная адресация.
9. Автоинкрементная и автодекрементная адресация.
10. Регистровая адресация.
11. Относительная адресация.
12. Команды обмена данными.
13. Арифметические команды.
14. Логические команды.
15. Команды сдвига и вращения.
16. Команды работы с флагами.
17. Команды безусловной передачи управления.
18. Команды сравнения, условного перехода и циклов.
19. Команды цепочек.
20. Команды работы со стеком.
21. Команды работы с битами.
22. Команды управления состоянием микропроцессора.
23. Команды работы с портами ввода-вывода.
24. Системные вызовы.
25. Организация массивов в Ассемблере.
26. Организация структур в Ассемблере.
27. Организация записей в Ассемблере.
28. Организация циклов в ассемблере.
29. Организация подпрограмм в Ассемблере.

Тема № 3

Комбинированные программы. Связывание разноязыковых модулей

Цель работы

Познакомиться с основными способами передачи параметров подпрограмм, особенностями передачи управления между модулями, научиться писать комбинированные программы, в которых модули Ассемблера вызываются из модулей, написанных на высокоуровневых языках программирования.

Краткие теоретические сведения

Процедуры на Ассемблере могут получать данные из вызывающей процедуры и могут возвращать или не возвращать ей результаты своей работы. Существует несколько способов передачи параметров в процедуры.

1. **Передача параметров через регистры**. Самый быстрый и самый простой способ. Используется, например, при вызове функций прерываний BIOS.

2. **Передача данных путем прямого обращения к памяти**. При таком способе обмена данными вызываемая и вызывающая процедуры обращаются напрямую к данным, описанным в любом месте (в том числе и в теле любой процедуры) программы.

3. **Передача параметров через таблицу адресов**. В этом случае в памяти вызывающей процедуры создается специальная таблица адресов параметров. В таблицу перед вызовом процедуры записывают адреса передаваемых данных. Затем адрес самой таблицы заносится в один из регистров и управление передается вызываемой процедуре. Вызываемая процедура сохраняет в стеке содержимое всех регистров, которые собирается использовать, после чего выбирает адреса переданных данных из таблицы, выполняет требуемые действия и заносит результат по адресу, переданному в той же таблице.

4. **Передача параметров в потоке кода**. При этом данные размещают прямо в коде программы, сразу после команды CALL. Чтобы прочитать параметр, процедура должна использовать его адрес, который автоматически передается в стеке как адрес возврата. В этом случае вызываемая процедура должна изменить адрес возврата на первый байт после конца передаваемых данных перед выполнением команды RET.

5. **Передача параметров через стек**. Это наиболее распространенный и надежный способ передачи. Именно его используют языки высокого уровня. Параметры помещают в стек командой PUSH, после чего управление передается вызываемой процедуре. Доступ к параметрам в стеке из вызываемой процедуры осуществляют через регистр BP, в который помещают адрес вершины стека, хранящийся в регистре указателя стека SP. Для обеспечения корректного возврата в вызывающую процедуру старое значение регистра BP помещают в стек первой командой процедуры. Параметры в стеке, адрес возврата и старое значение BP вместе называют активизационной записью процедуры. Вызываемая проце-

дура, зная структуру стека, извлекает параметры в соответствующие регистры, выполняет над ними операции и записывает результат, используя адрес, переданный в стеке.

Удаление параметров из стека можно организовать по-разному. Если стек освобождает вызываемая процедура по команде RET, то код программы получается более коротким. Если за освобождение стека отвечает вызывающая программа, то становится возможным вызов нескольких процедур с одними и теми же значениями параметров просто последовательными командами CALL.

Второй способ используется в языках Си и Си++ и дает больше возможностей для оптимизации. Вопрос о порядке записи параметров в стек для ассемблера не столь важен, так как и записывает и извлекает подпрограммы все сами. А вот при взаимодействии ассемблера с языками высокого уровня, следует знать особенности связи модулей этих языков. Рассмотрим эти особенности на примере языка Си.

Итак, передача параметров в Си осуществляется через стек. Причем, что именно помещается в стек (значение или адрес) определяется явно средствами языка. При передаче параметров Си руководствуется внутренним представлением данных. Параметры помещаются в стек в соответствии с прототипом в обратном порядке, то есть справа налево.

Так при вызове функции с прототипом:

```
void a(int p1, int p2, long int p3);
```

в стек сначала будет занесен параметр **p3** (длиной 4 байта), затем **p2** и **p1** (по два байта каждый), а затем уже адрес возврата (ближний или дальний в зависимости от используемой модели памяти).

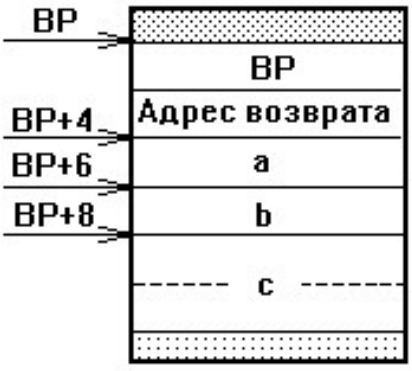

Рис. 3.1. Стек при передаче параметров

После вызова функции стек восстановит вызывающая программа.

Если используется функция с переменным числом параметров, то это отразится только на размере области параметров, так как каждый параметр будет помещен в стек, а удаление параметров будет выполнять вызывающая программа.

Возвращаемые значения должны быть записаны в регистры:

– **char, short, int, enum,** указатели **near -** в регистр AX;

– указатели **far, huge** и прочие 4-х байтовые величины - в регистры DX:AX;

– **float, double -** в регистры TOS и ST(0) сопроцессора;

– **struct -** записывается в память, а в регистр записывается указатель (структуры длиной в 1 и 2 байта возвращаются в AX, а 4 байта - в DX:AX).

Существует еще одна особенность внутреннего представления программ на языке Си++: компилятор языка изменяет используемые имена. Этот процесс, называемый обработкой имен, выполнят сохранение информации о типах аргументов функции, путем модификации ее имени таким образом, чтобы оно указывало на типы аргументов. При разработке программ на Си++ обработка имени выполняется автоматически и скрыта от программиста. Однако если какой-то модуль написан на Ассемблере, программист должен самостоятельно выполнить обработку имен функций в этом модуле. Для этого необходимо знать соглашения, принятые в языке Си++.

1. Перед глобальными именами ставится символ подчеркивания;

2. К именам функций в начало добавляется символ **@**, а в конец дописываются знаки **$q** и символы, кодирующие типы параметров функции в виде:

```
@ <имя функции> $q<коды типов параметров>
```

Таблица 3.1 - Коды типов параметров

Тип	Эквивалент	Тип	Эквивалент	Тип	Эквивалент
void	v	float	f	long	l
char	zc	double	d	*, [ ]	p
int	i	short	s	…	e

Например:

```
fa(int *s[], char c, short t) => @fa$qppizcs.
```

3. Для отмены чувствительности Си++ к регистру следует указать опцию Case sensitive …off.

Обработку имен ассемблерных функций можно и не выполнять, например, чтобы избежать несовместимости с последующими версиями компиляторов, в которых возможны изменения алгоритма этой обработки. С этой целью Си++ позволяет использовать стандартные имена функций Си в программах написанных на Си++, например:

```
Extern "C" { int SUM (int *a, int b) }
```

Все функции, объявление которых заключено в фигурные скобки, будут иметь имена, соответствующие соглашениям, принятым в языке Си. Приведенная выше функция на Ассемблере SUM будет иметь следующий вид:

```
public _SUM
_SUM proc
```

Таким образом, при объявлении в ассемблерном модуле функций, включенных в блок `extern "C"`, нет необходимости выполнять обработку их имен.

Для того чтобы скомпоновать модули на ассемблере с программой, написанной на Си, необходимо следовать определенным соглашениям.

При компиляции исходной программы на Си создаются следующие сегменты:

- сегмент кода;
- сегмент данных;
- сегмент неинициализированных данных.

Используемая модель памяти влияет не только на тип вызываемой функции и указателей на данные, но и на то, какие сегменты будет использоваться программой. В таблице 3.2 приведены имена сегментов, используемые Си для различных моделей памяти.

Таблица 3.2 - Имена сегментов, используемые различными моделями памяти

Модель памяти	**Сегмент кодов**	**Сегмент инициированных данных**	**Группа сегментов данных, адресуемых DS**
Tiny	_TEXT	_DATA	DGROUP
Small	_TEXT	_DATA	DGROUP
Compact	_TEXT	_DATA	DGROUP
Middle	<имя фала>_TEXT	_DATA	DGROUP
Large	<имя фала>_TEXT	_DATA	DGROUP
Hugo	<имя фала>_TEXT	<имя файла>_DATA	<имя файла>_DATA

Си позволяет ассемблеру увеличивать список глобальных переменных, доступных для всех модулей. Это достигается за счет размещения переменных в сегменте данных, отведенном для глобальных переменных, и описания его внутренним **public.** Имя такой переменной по правилам Си должно начинаться со знака подчеркивания. Прочие модули, использующие данное имя, должны включать его описание как **extrn** (на Ассемблере) или **extern** (на Си).

Приведем примеры программы.

Листинг 3.1.1 Определение минимального значения из двух заданных (реализация с переменным количеством параметров функции). С++:

```
#include <stdio.h>
extern int amin(int count,int v1,int v2,...); //первый параметр -
счетчик
void main()
{
  int a=3,b=5,c;
  c=amin(5,a,b,1,10,0);
  printf("c=%d",c);
}
```

Листинг 3.1.2. Тот же модуль на Ассемблере:

```
_TEXT segment byte public 'CODE'
  assume CS:_TEXT
  public @amin$qiiie
```

```
@amin$qiiie proc near
  pushBP
  mov BP,SP
  mov AX,0
  mov CX,[BP+4]  ; в CX заносится количество значений
  cmp CX,AX
  jle exit
  mov AX,[BP+6]  ; в AX заносится первое значение из списка
  jmp short ltest
compare:
  cmp AX,[BP+6]
  jle ltest
  mov AX,[BP+6]
ltest:
  add BP,2
  loopcompare
exit:
  pop BP
  ret
@amin$qiiie endp
_TEXT ends
end
```

Листинг 3.2.1. Определение среднего арифметического последовательности из 10 чисел. Си вызывает функцию на ассемблере для суммирования чисел, а ассемблер вызывает функцию на Си для выполнения операции деления в вещественной арифметике:

```
#include <stdio.h>
extern float Average(int far * ValuePtr, int NumberOfValues);

#define NUMBER_OF_TEST_VALUES 10
int TestValues[NUMBER_OF_TEST_VALUES] = {1,2,3,4,5,6,7,8,9,10};
main()
{
  printf("The average value is: %f\n",
  Average(TestValues, NUMBER_OF_TEST_VALUES));
}
float IntDivide(int Dividend, int Divisor)
{
  return( (float) Dividend / (float) Divisor );
}
```

Листинг 3.2.2. Тот же модуль на Ассемблере:

```
.MODEL  SMALL
  EXTRN   @IntDivide$qii:PROC
.CODE
  PUBLIC  @Average$qnii
@Average$qnii    PROC
```

```
  push BP
  mov BP,SP
  les BX,[BP+4]  ; загрузка в ES:BX адреса массива значений
  mov CX,[BP+8]  ; загрузка количества чисел
  mov AX,0       ; обнуление суммы
AverageLoop:
  add AX,ES:[BX] ; добавление очередного значения
  add BX,2       ; переход к следующему значению
  loop AverageLoop
  push WORD PTR [BP+8] ; количество чисел в стек (второй параметр)
  push AX              ; запись в стек суммы чисел (первый параметр)
  call @IntDivide$qii  ; вызов функции на Си
  add SP,4             ; удаление параметров
  pop BP
  ret                  ; среднее значение находится в регистре
@Average$qnii    ENDP
end
```

При подключении модуля на ассемблере в Visual C++, его необходимо предварительно откомпилировать. Затем полученный объектный модуль, в котором находится ассемблерная процедура, необходимо подключить к приложению в файл проекта следующим образом:

```
extern  "C" void __<конвенция>  ADD1(int a,intb,int &c);
```

Модуль на Ассемблере необходимо транслировать с опциями:

```
> ML /c /coff program.asm
```

или

```
> tasm /ml program.asm
```

Листинг 3.3.1. Пример использования конвенции stdcall:

```
extern  "C" void __stdcall  ADD1(int a,intb,int &c);
. 386
. model flat
. code
  public _ADD1
  _ADD1 proc
  push EBP
  push EBP,ESP
  mov EAX,dword ptr [EBP+8]
  add EAX,dword ptr [EBP+12]
  mov EDX,dword ptr [EBP+16]
  mov [EDX],EAX
  pop EBP
  ret 12   ;стек освобождает сама процедура
_ADD1 endp
```

```
end
```

Листинг 3.3.2. Пример использования конвенции fastcall:

```
extern "C" void __fastcall  ADD1(int a,intb,int &c);
. 386
. model flat
. code
public @ADD1@12
@ADD1@12 proc
  Add EAX,EDX
  mov [EDX],EAX
  ret            ;стек освобождает вызывающая программа
@ADD1@12 endp
end
```

Задание

1. Изучить краткие теоретические сведения, материалы лекций по теме практического занятия и приведенные выше примеры программ.
2. Получить индивидуальное задание у преподавателя, реализовать соответствующие модули на Си и Ассемблере и скомбинировать их в один исполняемый файл.
3. Написать отчет и защитить у преподавателя.

Варианты заданий

Оформить ассемблерный код, полученный в ходе работы над заданием №2, в процедуру, получить объектный модуль, подключить его в проект Visual C++ (или любой другой) и вызвать процедуру, передав входные данные одним из описанных способов.

Контрольные вопросы

1. Передача параметров через регистры.
2. Передача данных путем прямого обращения к памяти.
3. Передача параметров через таблицу адресов.
4. Передача параметров в потоке кода.
5. Передача параметров через стек.
6. Для чего предназначены команды CALL и RET в языке Ассемблер?
7. Соглашение для имен функция в C++.
8. Как можно подключить модуль, написанный на Ассемблере в C++?

Тема № 4

Процессы

Цель работы

Познакомиться с основными функциями WinAPI и POSIX API для работы с процессами, особенностями процессов в операционных системах Windows и Unix.

Краткие теоретические сведения

4.1. Процессы, общие сведения

Процесс можно рассматривать как программу на стадии выполнения, "объект", которому выделено процессорное время или как акт асинхронной работы.

Процесс может находиться в одном из состояний, представленных на рисунке 4.1.

Рис. 4.1. Состояния процесса (ОС UNIX)

Над процессами можно производить следующие операции:

1. Создание процесса - это переход из состояния рождения в состояние готовности.
2. Уничтожение процесса - это переход из состояния выполнения в состояние смерти.
3. Восстановление процесса - переход из состояния готовности в состояние выполнения.
4. Изменение приоритета процесса - переход из выполнения в готовность.

5. Блокирование процесса - переход в состояние ожидания из состояния выполнения.

6. Пробуждение процесса - переход из состояния ожидания в состояние готовности.

7. Запуск процесса (или его выбор) - переход из состояния готовности в состояние выполнения.

Для создания процесса операционной системе нужно:

1. Присвоить процессу имя.
2. Добавить информацию о процессе в список процессов.
3. Определить приоритет процесса.
4. Сформировать блок управления процессом.
5. Предоставить процессу нужные ему ресурсы.

4.2. Процессы в Windows

В Windows под процессом понимается объект ядра, которому принадлежат системные ресурсы, используемые приложением. Поэтому можно сказать, что в Windows процессом является приложение. Выполнение каждого процесса начинается с первичного потока. В процессе своего исполнения процесс может создавать другие потоки. Исполнение процесса заканчивается при завершении работы всех его потоков. Процесс может быть также завершен вызовом функций **ExitProcess** и **TerminateProcess**.

Новый процесс в Windows создается вызовом функции **CreateProcess**, которая имеет следующий прототип:

```
BOOL CreateProcess
(
  LPCTSTR lpApplicationName,  // имя исполняемого модуля
  LPTSTR lpCommandLine,       // командная строка
  LPSECURITY_ATTRIBUTES lpProcessAttributes,   // атрибуты защиты для
нового приложения
  LPSECURITY_ATTRIBUTES lpThreadAttributes,    // атрибуты защиты для
первого потока созданного приложением
  BOOL bInheritHandles, // Флаг наследования от процесса производящего
запуск
  DWORD dwCreationFlags, // Флаг способа создание процесса и его
приоритета
  LPVOID lpEnvironment, // Указатель на блок переменных окружения
  LPCTSTR lpCurrentDirectory, // Текущий диск или каталог
  LPSTARTUPINFO lpStartupInfo, // настройки свойств процесса, например
расположения окон и заголовок
  LPPROCESS_INFORMATION lpProcessInformation   // Указатель на структуру с
информацией о процессе.
);
```

Функция CreateProcess возвращает значение TRUE, если процесс был создан успешно. В противном случае эта функция возвращает значение FALSE. Процесс, который создает новый процесс, называется родительским процессом (par-

ent process) по отношению к создаваемому процессу. Новый же процесс, который создается другим процессом, называется дочерним процессом (child process) по отношению к процессу родителю

Первый параметр **lpApplicationName** определяет строку с именем exe-файла, который будет запускаться при создании нового процесса. Эта строка должна заканчиваться нулем и содержать полный путь к запускаемому файлу. Напишем простую программу, которая выводит на консоль свое имя и параметры (см. листинг 4.1).

Листинг 4.1. Пример простой консольной программы:

```
#include <conio.h>
int main(int argc, char *argv[])
{
 int i;
 _cputs("Я создался.");
 _cputs("\nМое имя: ");
 _cputs(argv[0]);
 for (i = 1; i < argc; i++)
 _cprintf ("\n Мой %d параметр = %s", i, argv[i]);
 _cputs("\nНажмите кнопку для выхода.\n");
 _getch();
 return 0;
}
```

Откомпилируем программу, полученный исполняемый файл назовем ConsoleProcess.exe и положим в корень диска C. Тогда этот exe-файл может быть запущен из другого приложения следующим образом (см. листинг 4.2). В программе создается процесс, который создает другое консольное приложение с новой консолью и ждет завершения работы этого приложения.

Листинг 4.2. Пример программы, создающей процесс:

```
#include <windows.h>
#include <conio.h>
int main()
{
 char lpszAppName[] = "C:\\ConsoleProcess.exe";
 STARTUPINFO si;
 PROCESS_INFORMATION piApp;
 ZeroMemory(&si, sizeof(STARTUPINFO));
 si.cb = sizeof(STARTUPINFO);

 // создаем новый консольный процесс
 if (!CreateProcess(lpszAppName, NULL, NULL, NULL, FALSE,
 CREATE_NEW_CONSOLE, NULL, NULL, &si, &piApp))
 {
 _cputs("Новый процесс не создан.\n");
 _cputs("Проверьте имя процесса.\n");
 _cputs("Нажмите кнопку для выхода.\n");
```

```
 _getch();
 return 0;
 }
 _cputs("Новый процесс создан.\n");

// ждем завершения созданного прцесса
 WaitForSingleObject(piApp.hProcess, INFINITE);

 // закрываем дескрипторы этого процесса в текущем процессе
 CloseHandle(piApp.hThread);
 CloseHandle(piApp.hProcess);
 return 0;
}
```

Обратите внимание, что перед запуском консольного процесса ConsoleProcess.exe все поля структуры si типа STARTUPINFO должны заполняться нулями. Это делается при помощи вызова функции **ZeroMemory**, которая предназначена для этой цели и имеет следующий прототип:

```
VOID ZeroMemory(
PVOID Destination, // адрес блока памяти
SIZE_T Length // длина блока памяти
);
```

В этом случае вид главного окна запускаемого приложения определяется по умолчанию самой операционной системой Windows.

В параметре **dwCreationFlags** устанавливается флаг CREATE_NEW_CONSOLE. Это говорит системе о том, что для нового создаваемого процесса должна быть создана новая консоль. Если этот параметр будет равен NULL, то новая консоль для запускаемого процесса не создается и весь консольный вывод нового процесса будет направляться в консоль родительского процесса.

Структура piApp типа PROCESS_INFORMATION содержит идентификаторы и дескрипторы нового создаваемого процесса и его главного потока. Мы не используем эти дескрипторы в нашей программе и поэтому закрываем их.

Значение FALSE параметра **bInheritHandle** говорит о том, что эти дескрипторы не являются наследуемыми.

Для ожидания завершения работы дочернего процесса используется функция WaitForSingleObject, имеющая следующий прототип:

```
DWORD WaitForSingleObject(
 HANDLE hHandle, // дескриптор объекта
 DWORD dwMilliseconds  // интервал ожидания в миллисекундах
);
```

Для бесконечного ожидания нужно установить второй параметр равным INFINITE.

Попробуем запустить новый консольный процесс другим способом, используя второй параметр функции CreateProcess – передадим системе имя нового процесса и его параметры через командную строку в параметре **lpCommandLine** (см. листинг 4.3).

Листинг 4.3. Пример программы, создающей процесс через lpCommandLine:

```
#include <windows.h>
#include <conio.h>
int main()
{
 char lpszCommandLine[] = "C:\\ConsoleProcess.exe p1 p2 p3";
 STARTUPINFO si;
 PROCESS_INFORMATION piCom;
 ZeroMemory(&si, sizeof(STARTUPINFO));
 si.cb = sizeof(STARTUPINFO);

 // создаем новый консольный процесс
 CreateProcess(NULL, lpszCommandLine, NULL, NULL, FALSE,
 CREATE_NEW_CONSOLE, NULL, NULL, &si, &piCom);

 Sleep(1000); // немного подождем и закончим свою работу

 // закрываем дескрипторыэтого процесса
 CloseHandle(piCom.hThread);
 CloseHandle(piCom.hProcess);
 _cputs("Новый процесс создан.\n");
 _cputs("Нажмите кнопку для выхода.\n"); 4
 _getch();
 return 0;
}
```

Процесс может завершить свою работу вызовом функции ExitProcess, которая имеет следующий прототип:

```
VOID ExitProcess(
UINT uExitCode // код возврата для всех потоков
);
```

При вызове функции **ExitProcess** завершаются все потоки процесса с кодом возврата, который является параметром этой функции. Приведем пример программы, которая завершает свою работу вызовом функции ExitProcess (см. листинг 4.4).

Листинг 4.4. Пример завершения процесса функцией ExitProcess:

```
#include <windows.h>
#include <iostream>
using namespace std;
volatile UINT count;
```

```
volatile char c;
void thread()
{
 for ( ; ; )
 {
 count++;
 Sleep(100);
 }
}
int main()
{
 HANDLE hThread;
 DWORD IDThread;
 hThread = CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)thread,
NULL, 0, &IDThread);
 if (hThread == NULL)
 return GetLastError();
 for ( ; ; )
 {
 cout << "Нажмите 'y' для вывода или 'e' для выхода: ";
 cin >> (char)c;
 if (c == 'y')
 cout << "count = " << count << endl;
 if (c == 'e')
 ExitProcess(1);
 }
}
```

Один процесс может завершить другой процесс при помощи вызова функции **TerminateProcess**, которая имеет следующий прототип:

```
 BOOL TerminateProcess(
 HANDLE hProcess,
 UINT uExitCode
 );
```

Если функция TerminateProcess выполнилась успешно, то она возвращает значение TRUE. В противном случае возвращаемое значение равно FALSE. Функция TerminateProcess завершает работу процесса, но не освобождает все ресурсы, принадлежащие этому процессу. Поэтому эта функция должна вызываться только в аварийных ситуациях при зависании процесса.

Приведем программу, которая демонстрируют работу функции TerminateProcess. Для этого сначала создадим бесконечный процесс-счетчик, который назовем ConsoleProcess.exe (см. листинг 4.5) и собственно программу (см. листинг 4.6), которая создает этот процесс, а потом завершает его по требованию пользователя.

Листинг 4.5. Пример бесконечного цикла:

```
#include <windows.h>
#include <iostream>
using namespace std;
int count=0;
void main()
{
 for ( ; ; )
 {
  count++;
  Sleep(1000);
  cout << "count = " << count << endl;
 }
}
```

Листинг 4.6. Пример использования TerminateProcess:

```
#include <windows.h>
#include <conio.h>
int main()
{
 char lpszAppName[] = "C:\\ConsoleProcess.exe";
 STARTUPINFO si;
 PROCESS_INFORMATION pi;
 ZeroMemory(&si, sizeof(STARTUPINFO));
 si.cb=sizeof(STARTUPINFO);

// создаем новый консольный процесс
 if (!CreateProcess(lpszAppName, NULL, NULL, NULL, FALSE,
 CREATE_NEW_CONSOLE, NULL, NULL, &si, &pi))
 {
  _cputs("Новый процесс не создан.\n");
  _cputs("Проверьте имя процесса.\n");
  _cputs("Нажмите кнопку для выхода.\n");
  _getch();
  return 0;
 }
 _cputs("Новый процесс создан.\n");
 while (true)
 {
  char c;
  _cputs("Нажмите 't' чтобы убить процесс: ");
  c = _getch();
  if (c == 't')
  {
   _cputs("t\n");

   // завершаем новый процесс
   TerminateProcess(pi.hProcess,1);
   break;
  }
```

```
    }
    // закрываем дескрипторы нового процесса в текущем процессе
    CloseHandle(pi.hThread);
    CloseHandle(pi.hProcess);
    return 0;
}
```

4.3. Процессы в Unix

В операционной системе Unix процесс не может взяться из ниоткуда - его обязательно должен запустить какой-то процесс. Процесс, запущенный другим процессом, называется дочерним (child) процессом или потомком. Процесс, который запустил процесс называется родительским (parent), родителем или просто - предком. Таким образом, процессы создают иерархию в виде дерева.

Самым "главным" предком, то есть процессом, стоящим на вершине этого дерева, является процесс init.

У каждого процесса есть два атрибута - PID (Process ID) - идентификатор процесса и PPID (Parent Process ID) - идентификатор родительского процесса. PID процесса init имеет значение 1.

Прежде чем приступить к программированию, рассмотрим несколько полезных команд операционной системы, предназначенных для работы с процессами.

Список процессов можно посмотреть командой **ps**.

Список процессов в реальном времени с сортировкой по степени нагрузки на процессор – командой **top**.

Убить процесс можно командой **kill**.

Посмотреть окружение процесса можно командой **printenv [имя],** а также **set –p**.

Изменить переменные окружения можно командой **export [имя[=значение]] ... [имя[=значение]].**

Посмотреть значение переменно окружения можно командой **echo $имя.**

Послать сигнал процессу **kill -sig pid.**

Команда **nice [-###] команда [аргументы]** позволяет запустить процесс с пониженным или повышенным приоритетом. Повысить приоритет команды может только пользователь root, указав соответствующий коэффициент понижения. Для увеличения приоритета нужно указать отрицательный коэффициент, например, **nice -5 process.**

Рассмотрим вывод команды top

```
[root@srv ~]# top
top - 05:15:45 up 44 days, 3:44, 2 users, load average: 0.16, 0.14, 0.10
Tasks: 189 total,   1 running, 188 sleeping,   0 stopped,   0 zombie
Cpu(s):  1.4%us,  0.2%sy,  0.0%ni, 98.4%id,  0.0%wa,  0.0%hi,  0.0%si,
0.0%st
Mem:   8165732k total,  7936848k used,   228884k free,   282992k buffers
Swap:  1052248k total,      828k used,  1051420k free,  6663884k cached

PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
17279 apache    16   0  270m  57m  35m S  5.0  0.7   0:16.27 httpd
```

```
 3065 mysql     15   0 1871m 257m 3996 S  0.7  3.2 451:09.00 mysqld
17348 apache    15   0  270m  54m  33m S  0.7  0.7   0:16.89 httpd
 3269 root      15   0  358m 6568 1796 S  0.3  0.1 397:17.15 fail2ban
 3271 root      15   0 13196 1352  972 S  0.3  0.0  45:19.98 gam_server
17273 apache    15   0  270m  54m  32m S  0.3  0.7   0:15.65 httpd
28027 nginx     15   0  159m 9760 2188 S  0.3  0.1  13:13.93 nginx
    1 root      15   0 10364  680  572 S  0.0  0.0   0:00.97 init
```

Рис. 4.2. Вывод команды top

В первой строке программа сообщает текущее время, время работы системы (44 days), количество зарегистрированных пользователей (2 users), общая средняя загрузка системы (load average) - среднее число процессов, находящихся в состоянии выполнения (R) или в состоянии ожидания (D). Общая средняя загрузка измеряется каждые 1, 5 и 15 минут.

Во второй строке вывода программы top сообщается, что в списке процессов находятся 189 процессов, из них 188 спят (находятся в состоянии готовности или ожидания), 1 выполняется, 0 процессов зомби и 0 остановленных процессов.

В третьей-пятой строках приводится информация о загрузке процессора, использования памяти и файла подкачки.

Далее отображается таблица процессов: PID (идентификатор процесса), USER (пользователь, запустивший процесс), STAT (состояние процесса) и COMMAND (команда, которая была введена для запуска процесса).

Колонка STAT может содержать следующие значения:

– R - процесс выполняется или готов к выполнению (состояние готовности).

– D - процесс в спящем неактивном состоянии, например, ожидает дискового ввода/вывода.

– T - процесс остановлен (stopped) или трассируется отладчиком.

– S - процесс в состоянии ожидания (sleeping).

– Z - процесс-зомби, т.е. процесс завершился, но его структура из списка процессов не удалена.

– < - процесс с отрицательным значением nice.

– N - процесс с положительным значением nice.

Рассмотрим основные системные вызовы, использующиеся в UNIX системах для работы с процессами.

Новый процесс можно породить с помощью системного вызова **fork()**. Синтаксис вызова следующий:

```
#include <sys/types>
#include <unistd.h>
pid_t fork(void);
```

pid_t является примитивным типом данных, который определяет идентификатор процесса или группы процессов. При вызове fork() порождается новый процесс (процесс-потомок), который почти идентичен порождающему процессу-родителю. При вызове fork() возникают два полностью идентичных процесса. Весь код после fork() выполняется дважды, как в процессе-потомке, так и в про-

цессе-родителе. Процесс-потомок и процесс-родитель получают разные коды возврата после вызова fork(). Процесс-родитель получает идентификатор (PID) потомка. Если это значение будет отрицательным, следовательно, при порождении процесса произошла ошибка. Процесс-потомок получает в качестве кода возврата значение 0, если вызов fork() оказался успешным.

Таким образом, можно проверить, был ли создан новый процесс:

```
switch(ret=fork())
{
  case -1: /{*}при вызове fork() возникла ошибка{*}/
  case 0 : /{*}это код потомка{*}/
  default : /{*}это код родительского процесса{*}/
}
```

Приведем пример порождения процесса через fork().

Листинг 4.7. Пример порождения процесса с помощью fork():

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
main()
{
  pid_t pid;
  int rv;
  switch(pid=fork()) {
  case -1:
    perror("fork"); /* произошла ошибка */
    exit(1); /*выход из родительского процесса*/
  case 0:
    printf(" CHILD: Это процесс-потомок!\n");
    printf(" CHILD: Мой PID -- %d\n", getpid());
    printf(" CHILD: PID моего родителя -- %d\n", getppid());
    printf(" CHILD: Введите мой код возврата (как можно меньше):");
    scanf(" %d");
    printf(" CHILD: Выход!\n");
    exit(rv);
  default:
    printf("PARENT: Это процесс-родитель!\n");
    printf("PARENT: Мой PID -- %d\n", getpid());
    printf("PARENT: PID моего потомка %d\n",pid);
    printf("PARENT: Я жду, пока потомок не вызовет exit()...\n");
    wait();
    printf("PARENT: Код возврата потомка:%d\n", WEXITSTATUS(rv));
    printf("PARENT: Выход!\n");
  }
```

```
}
```

Родительскому процессу необходимо обмениваться информацией с дочерними или хотя бы синхронизироваться с ними, чтобы выполнять операции в нужное время. Один из способов синхронизации процессов – системные вызовы **wait()** и **waitpid()**.

Когда потомок вызывает **exit()**, код возврата передается родителю, который ожидает его, вызывая **wait()**. WEXITSTATUS() представляет собой макрос, который получает фактический код возврата потомка из вызова wait().

Функция **wait():**

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *status)
```

приостанавливает выполнение текущего процесса до завершения какого-либо из его процессов-потомков.

Иногда необходимо точно определить, какой из потомков должен завершиться. Для этого используется вызов **waitpid()** с соответствующим PID потомка в качестве аргумента – функция приостанавливает выполнение текущего процесса до завершения заданного процесса или проверяет завершение заданного процесса:

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t waitpid (pid_t pid, int *status, int options)
```

Еще один момент, на который следует обратить внимание при анализе примера, это то, что и родитель, и потомок используют переменную rv. Это не означает, что переменная разделена между процессами. Каждый процесс содержит собственные копии всех переменных.

Откомпилируем программу (см. листинг 4.7), сделаем файл исполняемым и запустим:

```
[root@srv ~]# gcc -o process process.c
[root@srv ~]# chmod +x ./process
[root@srv ~]# ./process
```

Перейдя на другую консоль (ALT + Fn) и введя команду

```
ps -a | grep process.
```

можно увидеть следующий вывод команды ps:

```
 4445 pts/1    00:00:15 process
```

Данный вывод означает, что нашему процессу присвоен идентификатор процесса 4445.

Так называемые **процессы-зомби** возникают, если потомок завершился, а родительский процесс не вызвал wait(). Для завершения процессов используют либо оператор возврата, либо вызов функции exit() со значением, которое нужно возвратить операционной системе. Операционная система оставляет процесс зарегистрированным в своей внутренней таблице данных, пока родительский процесс не получит кода возврата потомка, либо не закончится сам. В случае процесса-зомби его код возврата не передается родителю, и запись об этом процессе не удаляется из таблицы процессов операционной системы. При дальнейшей работе и появлении новых зомби таблица процессов может быть заполнена, что приведет к невозможности создания новых процессов.

Напишем программу (см. листинг 4.8), порождающую зомби, который будет существовать 8 секунд. Процесс-родитель будет ожидать завершения процесса-потомка через 15 секунд, а процесс-потомок завершится через 2 секунды.

Листинг 4.8. Пример порождения процесса-зомби:

```
#include <unistd.h>
#include <signal.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <stdio.h>

int main() {
   int pid;
   int status, died;

   pid=fork();
   switch(pid) {
   case -1: printf("can't fork\n");
            exit(-1);
   case 0 : printf("   I'm the child of PID %d\n", getppid());
            printf("   My PID is %d\n", getpid());
        // Ждем 2 секунды и завершаемся
         //sleep(2); // чтобы зомби "прожил" на 2 секунды больше

            exit(0);
   default: printf("I'm the parent.\n");
            printf("My PID is %d\n", getpid());
     // Ждем завершения дочернего процесса через 15 секунд, а потом
убиваем его
     sleep(15);
            if (pid & 1)
               kill(pid,SIGKILL);
            died= wait(&status);
  }
}
```

Программа выведет следующую информацию:

```
[root@srv ~]# ./zombie
I'm the parent
  My PID is 1431
  I'm the child of PID 1431
  My PID is 1432
```

Запомните последний номер и быстро переключайтесь на другую консоль (сочетание Alt+F2). Затем введите команду top –p 1432:

```
[root@srv ~]# top -p 1432
16:04:22  up 2 min,  3 users,  load average: 0,10, 0,10, 0,04
1 processes: 0 sleeping, 0 running, 1 zombie, 0 stopped
CPU states:  4,5% user,  7,6% system,  0,0% nice,  0,0% iowait,
87,8% idle
Mem:   127560k av,   76992k used,   50568k free,       0k shrd,
3872k buff
        24280k active,               19328k inactive
Swap:  152576k av,       0k used,  152576k free
39704k cached

  PID USER     PRI  NI  SIZE  RSS SHARE STAT %CPU %MEM   TIME
COMMAND
 1148 den       17   0     0    0     0 Z     0,0  0,0   0:00 zombie
<defunct>
```

Мы видим, что в списке процессов появился 1 зомби (STAT=Z), который проживет 10 секунд.

Для изменения пользовательского контекста процесса применяется системный вызов **exec(),** который пользователь не может вызвать непосредственно. Вызов **exec()** заменяет пользовательский контекст текущего процесса на содержимое некоторого исполняемого файла и устанавливает начальные значения регистров процессора. Этот вызов требует для своей работы задания имени исполняемого файла, аргументов командной строки и параметров окружающей среды. Для осуществления вызова программист может воспользоваться одной из шести функций: **execlp(), execvp(), execl(), execv(), execle(), execve(),** отличающихся друг от друга представлением параметров, необходимых для работы системного вызова **exec()**.

Прототипы функций:

```
#include <unistd.h>
int execlp(const char *file, const char *arg0,… const char *argN,
(char *)NULL );

int execvp(const char *file, char *argv[]);

int execl(const char *path, const char *arg0,… const char *argN,
(char *)NULL );
```

```
int execv(const char *path, char *argv[]);

int execle(const char *path, const char *arg0,… const char *argN,
(char *)NULL, char *envp[]);

int execve(const char *path, char *argv[], char *envp[]);
```

file - указатель на имя файла, который должен быть загружен.
path - указатель на полный путь к файлу, который должен быть загружен.
arg0,… ,argN - указатели на аргументы командной строки.
argv - массив из указателей на аргументы командной строки.
envp - массив указателей на параметры окружающей среды.

Суффиксы l, v, p и e, добавляемые к имени семейства exec обозначают, что данная функция будет работать с некоторыми особенностями:

p - определяет, что функция будет искать «дочернюю» программу в директориях, определяемых переменной среды PATH. Без суффикса p поиск будет производиться только в рабочем каталоге. Если параметр path не содержит маршрута, то поиск производится в текущей директории, а затем по маршрутам, определяемым переменной окружения PATH.

l - показывает, что адресные указатели (arg0, arg1,..., argn) передаются, как отдельные аргументы. Обычно суффикс l употребляется, когда число передаваемых аргументов заранее вам известно.

v - показывает, что адресные указатели (arg[0], arg[1],...arg[n]) передаются, как массив указателей. Обычно, суффикс v используется, когда передается переменное число аргументов.

e - показывает, что «дочернему» процессу может быть передан аргумент envp, который позволяет выбирать среду «дочернего» процесса. Без суффикса e «дочерний» процесс унаследует среду«родительского» процесса.

В случае успешного выполнения возврата из функций в программу, осуществившую вызов, не происходит, а управление передается загруженной программе. В случае неудачного выполнения в программу, инициировавшую вызов, возвращается отрицательное значение.

Создадим две программы, первая из которых просто печатает сообщение, а вторая взывает execl() для замены контекста дочернего процесса (см. листинг 4.9).

Листинг 4.9. Пример использования системного вызова exec():

```
// листинг 1ой программы
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

int main(int argc, char *argv[], char *enpv[])
{
 printf("Second program");
 return 0;
}
```

```
// листинг 2ой программы
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

int main(int argc, char *argv[], char *enpv[])
{
  pid_t num;
  num = fork(); // порождаем новый процесс.

  if(num == 0)
  {
   execl("2program",NULL,NULL);
 /* если дочерний процесс, то заменяем контекст дочернего процесса
и теперь запустилась 2-я программа */
  }
  else
  if(num > 0)
  {
    printf("Parent process\n\n");
  }
return 0;
}
```

Для изменения приоритетов порожденных процессов используются функции **setpriority()** и **getpriority()** (см. листинг 2.10). Приоритеты задаются в диапазоне от -20 (высший) до 20 (низший), нормальное значение - 0. Заметим, что повысить приоритет выше нормального может только суперпользователь (root).

Листинг 2.10. Пример использования системных вызовов setpriority() и getpriority():

```
#include <sys/time.h>
#include <sys/resource.h>
int process( int i)
{
  setpriority(PRIO_PROCESS, getpid(),i);
  printf("Process %d ThreadID: %d working with priority %d\n",i,
getpid(),getpriority(PRIO_PROCESS,getpid()));

  return(getpriority(PRIO_PROCESS,getpid()));
}
```

Для уничтожения процесса служит системный вызов **kill()**:

```
#include <sys/types.h>
#include <signal.h>
int kill(pid_t pid, int sig);
```

Если pid больше 0, то он задает PID процесса, которому посылается сигнал. Если pid равен 0, то сигнал посылается всем процессам той группы, к которой принадлежит текущий процесс.

sig - тип сигнала. Некоторые типы сигналов в Linux:

SIGKILL - сигнал приводит к немедленному завершению процесса. Этот сигнал процесс не может игнорировать.

SIGTERM - сигнал является запросом на завершение процесса.

SIGCHLD - система посылает этот сигнал процессу при завершении одного из его дочерних процессов.

Листинг 4.11. Пример использования системного вызова kill():

```
if (pid[i] == status)
{
  printf("ThreadID: %d finished with status %d\n", pid[i],
WEXITSTATUS(status));
}
else kill(pid[i],SIGKILL);
```

Задание

1. Изучить краткие теоретические сведения, материалы лекций по теме практического занятия и приведенные выше примеры программ.
2. Реализовать программы для ОС Windows, в которых используются функции CreateProcess, ExitProcess, TerminateProcess, OpenProcess, GetProcessId и др. и продемонстрировать их работу.
3. Реализовать программы для ОС Unix, в которых используются системные вызовы fork(), exec(), wait(), exit(), kill() и др. и продемонстрировать их работу.
4. Научиться использовать команды top, ps, kill, nice, export, set и изучить их параметры.
5. Написать отчет и защитить у преподавателя.

Контрольные вопросы

1. Что такое процесс?
2. Что такое многозадачность? Какими способами можно достичь многозадачности? По каким критериям оценивается эффективность многозадачности?
3. Что входит в понятие «контекст процесса»?
4. Что такое дескриптор процесса?
5. Какие события могут привести к созданию процесса и завершению процесса?
6. В каких состояниях может находиться процесс?
7. Что такое приоритет процесса? Как можно изменить приоритет процесса?
8. Какие особенности создания процесса в Unix Вам известны?

9. Что такое процесс-зомби и процесс-сирота? Каким образом они могут возникнуть?

10. Какие API функции Вам известны для работы с процессами в ОС Windows?

11. Какие API функции Вам известны для работы с процессами в ОС Unix/Linux?

12. Какими командами можно управлять процессами в ОС Unix/Linux?

13. С помощью каких средств можно управлять процессами в ОС Windows?

Варианты заданий

1. Для Windows и Linux написать по две программы. Первая реализует алгоритм поиска простых чисел в некотором интервале. Вторая - разбивает заданный интервал на диапазоны, осуществляет поиск простых чисел в каждом из интервалов в отдельном процессе, выводит общий результат.

2. Для Windows и Linux написать по две программы. Первая реализует алгоритм поиска указанной подстроки в строке. Вторая - разбивает входной файл на фрагменты, осуществляет поиск подстроки в каждом из фрагментов в отдельном процессе, выводит общий результат.

3. Для Windows и Linux написать по две программы. Первая реализует алгоритм умножения двух векторов произвольной длины. Вторая - умножает матрицу произвольного порядка на вектор, при этом умножение каждой строки на вектор производить в отдельном процессе.

4. Для Windows и Linux написать по две программы, одна из которых осуществляет проверку доступности узла сети на доступность (команда ping), а вторая - осуществляет проверку доступности диапазона IP-адресов сети класса «С» (254 адреса, маска 255.255.255.0), разделенного на несколько поддиапазонов, с помощью первой программы.

5. Координаты заданного количества шариков изменяются на случайную величину по вертикали и горизонтали, при выпадении шарика за нижнюю границу допустимой области шарик исчезает. Напишите программу (для Windows и Linux) изменения координат одного шарика и программу (для Windows и Linux), создающую для каждого из заданного количества шариков порожденный процесс изменения их координат.

6. Противостояние двух команд – каждая команда увеличивается на случайное количество бойцов и убивает случайное количество бойцов участника. Напишите программу (для Windows и Linux), которая бы осуществляла уменьшение числа бойцов в противостоящей команде и увеличение в своей на случайную величину и программу (для Windows и Linux), в которой бы в родительском процессе запускались порожденные процессы, реализующие деятельность одной команды.

7. Для Windows и Linux написать по две программы. Первая - копирует файл. Вторая - копирует содержимое директории пофайлово с помощью первой программы в отдельных процессах.

8. Для Windows и Linux написать по две программы. Первая – вычисляет контрольную сумму файла. Вторая – вычисляет контрольную сумму всех файлов в директории, при этом обработка каждого отдельного файла осуществляется с помощью первой программы в отдельном процессе.

9. Для Windows и Linux написать по две программы. Первая – вычисляет математическое ожидание и дисперсию в массиве данных. Вторая – вычисляет математическое ожидание и дисперсию в нескольких массивах данных, при этом обработка каждого массива осуществляется отдельно с помощью первой программы в новом процессе.

10. Для Windows и Linux написать по две программы. Первая – вычисляет частоты встречаемости в тексте биграмм символов (аа, аб, ав, … яэ, яю, яя). Вторая – принимает на вход текст, делит его на отдельные фрагменты и вычисляет частоты встречаемости в тексте биграмм символов, путем вызова первой программы для отдельных фрагментов текста.

11. Для Windows и Linux написать по две программы. Первая – вычисляет частоты встречаемости в тексте триграмм символов (ааа, ааб, аав, … ияэ, яяю, яяя). Вторая – принимает на вход текст, делит его на отдельные фрагменты и вычисляет частоты встречаемости в тексте триграмм символов, путем вызова первой программы для отдельных фрагментов текста.

12. Для Windows и Linux написать по две программы. Первая – вычисляет частоты встречаемости в тексте биграмм слов. Например, для предыдущего предложения биграммы слов это пары «первая вычисляет», «вычисляет частоты», «частоты встречаемости», «встречаемости в» и т.д. Вторая – принимает на вход текст, делит его на отдельные фрагменты и вычисляет частоты встречаемости в тексте биграмм слов, путем вызова первой программы для отдельных фрагментов текста.

13. Для Windows и Linux написать по две программы. Первая – вычисляет частоты встречаемости в тексте триграмм слов. Например, для предыдущего предложения биграммы слов это пары «первая вычисляет частоты», «вычисляет частоты встречаемости», «частоты встречаемости в», «встречаемости в тексте» и т.д. Вторая – принимает на вход текст, делит его на отдельные фрагменты и вычисляет частоты встречаемости в тексте триграмм слов, путем вызова первой программы для отдельных фрагментов текста.

14. Медведь и пчелы. Заданное количество пчел добывают мед равными порциями, задерживаясь в пути на случайное время. Медведь потребляет мед порциями заданной величины за заданное время и столько же времени может прожить без питания. Для Windows и Linux написать по две программы. Первая - реализует работу одной пчелы. Вторая - осуществляет

работу медведя, при этому для заданного количества пчел вызывается отдельный процесс работы одной пчелы.

15. Авиаразведка - создается условная карта в виде матрицы, размерность которой определяет размер карты, содержащей произвольное количество единиц (целей) в произвольных ячейках. Из произвольной точки карты стартуют несколько разведчиков (процессов), курсы которых выбираются так, чтобы покрыть максимальную площадь карты. Каждый разведчик фиксирует цели, чьи координаты совпадают с его координатами и по достижении границ карты сообщает количество обнаруженных целей. Реализуйте соответствующие программы для Windows и Linux, используя механизмы процессов.

16. Бег с препятствиями - создается условная карта трассы в виде матрицы, ширина которой соответствует количеству бегунов, а высота – фиксирована, содержащей произвольное количество единиц (препятствий) в произвольных ячейках. Стартующие бегуны (процессы) перемещаются по трассе и при встрече с препятствием задерживаются на фиксированное время. По достижении финиша бегуны сообщают свой номер. Реализуйте соответствующие программы для Windows и Linux, используя механизмы процессов.

Тема № 5

Потоки

Цель работы

Изучить работу с потоками. Научиться разбивать задачу на части, для последующего их выполнения различными потоками. Познакомиться с основными функциями WinAPI для работы с потоками в Windows и библиотекой Pthread для работы с потоками в Linux.

Краткие теоретические сведения

5.1. Потоки, общие сведения

Многопоточность является естественным продолжением многозадачности и представляет собой логическое развитие концепции разделения ресурсов. Потоки предоставляют возможность проведения параллельных или псевдопараллельных, в случае одного процессора, вычислений. Потоки могут порождаться во время работы программы, процесса или другого потока. Путь выполнения потока задается при его создании, указанием его стартовой функции, созданный поток начинает выполнять команды этой функции, и завершается когда происходит возврат из функции. Таким образом, несколько порожденных в программе потоков, могут пользоваться глобальными переменными, и любое изменение данных одним потоком, будет доступно и для всех остальных.

Основные отличия процесса от потока заключаются в том, что, каждому процессу соответствует своя независимая от других область памяти, таблица открытых файлов, текущая директория и прочая информация уровня ядра. Потоки же не связаны непосредственно с этими сущностями. У всех потоков принадлежащих данному процессу всё выше перечисленное общее, поскольку принадлежит этому процессу. Кроме того, процесс всегда является сущностью уровня ядра, то есть ядро знает о его существовании, в то время как потоки зачастую является сущностями уровня пользователя и ядро может ничего не знать о ней. В подобных реализациях все данные о потоке хранятся в пользовательской области памяти, и соответственно такие процедуры как порождение или переключение между потоками не требуют обращения к ядру и занимают на порядок меньше времени.

Потоки часто становятся источниками программных ошибок особого рода. Эти ошибки возникают при использовании потоками разделяемых ресурсов системы и являются частным случаем более широкого класса ошибок – ошибок синхронизации. Если задача разделена между независимыми процессами, то доступом к их общим ресурсам управляет операционная система, и вероятность ошибок из-за конфликтов доступа снижается. В

пользу потоков можно указать то, что накладные расходы на создание нового потока в многопоточном приложении ниже, чем накладные расходы на создание нового самостоятельного процесса. Уровень контроля над потоками в многопоточном приложении выше, чем уровень контроля приложения над дочерними процессами. Кроме того, многопоточные программы не склонны оставлять за собой процессы-зомби или независимые процессы без процесса-родителя.

5.2. Модели построения многопоточных приложений

Существует несколько моделей построения многопоточных приложений:

1. **Итеративный параллелизм** используется для реализации нескольких потоков (часто идентичных), каждый из которых содержит циклы. Потоки программы, описываются итеративными функциями и работают совместно над решением одной задачи.

2. **Рекурсивный параллелизм** используется в программах с одной или несколькими рекурсивными процедурами, вызов которых независим. Это технология «разделяй-и-властвуй» или «перебор-с-возвратами».

3. **Производители и потребители** – это парадигма взаимодействующих неравноправных потоков. Одни из потоков «производят» данные, другие их «потребляют». Часто такие потоки организуются в конвейер, через который проходит информация. Каждый поток конвейера потребляет выход своего предшественника и производит входные данные для своего последователя. Другой распространенный способ организации потоков – древовидная структура или сети слияния, на этом основан, в частности, принцип дихотомии.

4. **Клиенты и серверы** – еще один способ взаимодействия неравноправных потоков. Клиентский поток запрашивает сервер и ждет ответа. Серверный поток ожидает запроса от клиента, затем действует в соответствии с поступившим запросом.

5. **Управляющий и рабочие** – модель организации вычислений, при которой существует поток, координирующий работу всех остальных потоков. Как правило, управляющий поток распределяет данные, собирает и анализирует результаты.

6. **Взаимодействующие равные** – модель, в которой исключен не занимающийся непосредственными вычислениями управляющий поток. Распределение работ в таком приложении либо фиксировано заранее, либо динамически определяется во время выполнения. Одним из распространенных способов динамического распределения работ является «портфель задач». Портфель задач, как правило, реализуется с помощью разделяемой переменной, доступ к которой в один момент времени имеет только один процесс.

5.3. Потоки в Windows

В среде Microsoft Windows создание процесса производится с помощью системной функции ***CreateProcess***. Выполнение этой функции приводит к порождению потока, который называют главным потоком процесса. Главный поток присутствует в любом процессе и часто остается единственным. Остальные потоки

могут создаваться в коде главного потока по усмотрению программиста с помощью функции WIN API ***CreateThread***. Таким образом, **процесс – это контейнер для потоков**. Процесс-контейнер содержит как минимум один поток и если потоков в процессе несколько, приложение (процесс) становится многопоточным. Поток, выполнивший функцию CreateThread называют родительским потоком по отношению к созданному им, а созданный поток - дочерним потоком.

Поток в ОС Windows является основным элементом выполнения любого приложения. Рассмотрим основные функции для работы с потоками.

Поток создается с помощью функции CreateThread:

```
HANDLE CreateThread (
  LPSECURITY_ATTRIBUTES lpThreadAttributes,//атрибуты защиты
  DWORD dwStackSize,                       //размер стека
  LPTHREAD_START_ROUTINE lpStartAddress,   //функция потока
  LPVOID lpThreadParameter,                //параметр функции потока
  DWORD dwCreationFlags,                   //параметр запуска потока
  LPDWORD lpThreadID                       //идентификатор потока
);
```

Данная функция создает дочерний поток, устанавливает его характеристики (атрибуты защиты, размер стека), указывает на код функции потока и в зависимости от значения параметра dwCreationFlags либо производит его запуск, либо приостанавливает до специального распоряжения. Под функцией потока понимают описанную в программе функцию, которая содержит код, выполнение которого должно осуществляться в рамках данного потока.

Параметры функции CreateThread (in - входные, out- выходные):

lpThreadAttributes – входной параметр, указатель на структуру SECURITY_ATTRIBUTES, определяющую атрибуты защиты для создаваемого потока. Рекомендуется задавать значение NULL, разрешающее использовать любые функции для управления данным потоком.

dwStackSize – входной параметр, размер стека потока в байтах. Для использования размера стека родительского потока используйте значение 0.

lpStartAddress – входной параметр, указатель на функцию программы, которую будет выполнять поток (можно задать просто имя этой функции). Функция потока принимает в качестве параметра единственный параметр и возвращает код завершения типа DWORD. Поток может представить этот параметр как значение типа DWORD или как указатель.

lpThreadParameter – входной параметр, указатель, который передается потоку в качестве параметра и обычно интерпретируется им, как указатель на некоторую структуру. При отсутствии параметров следует указать NULL.

dwCreationFlags – входной параметр запуска потока. Нулевое значение параметра означает, что поток готов к немедленному выполнению. Если в качестве значение этого параметра указать константу CREATE_SUSPENDED, то новый поток будет находиться в состоянии ожидания до тех пор, пока не будет вызвана функция ResumeThread.

lpIDThread – выходной параметр, указатель на переменную типа DWORD, в которую будет помещен идентификатор (системный номер) созданного потока.

Возвращаемое значение: в случае успеха функция CreateThread возвращает дескриптор созданного потока (тип handle), который необходим для выполнения различных операций над потоком. При ошибке функция возвращает значение NULL.

Поток может завершиться по собственной инициативе или по инициативе родительского потока, формируя при этом код завершения.

В первом случае поток завершается при выполнении оператора возврата из функции потока (return) или с помощью функции ***ExitThread***:

```
VOID ExitThread(
   DWORD  dwExitCode  // код завершения потока
);
```

В качестве единственного параметра этой функции задается код завершения потока.

Во втором случае применяется функция TerminateThread, с помощью которой родительский поток может принудительно завершить выполнение своего дочернего потока:

```
BOOL TerminateThread(
 HANDLE  hThread,          // дескриптор потока
 DWORD  dwExitCode    // код завершения потока
);
```

Значение дескриптора потока определяется по значению, возвращаемому функцией CreateThread при создании потока.

Все потоки, созданные в рамках какого-либо процесса, автоматически завершают свое выполнение при завершении работы процесса (т.е. выполнении функции ExitProcess).

Для получения кода завершения ранее запущенного дочернего потока используется функция ***GetExitCodeThread***:

```
BOOL GetExitCodeThread(
 HANDLE  hThread,          // дескриптор потока
 LPDWORD lpdwExitCode // адрес для приема кода завершения
);
```

Если поток, для которого вызвана данная функция, все еще работает, вместо кода завершения возвращается значение STILL_ACTIVE.

Для перевода родительского потока в режим ожидания (блокирования) до момента завершения нескольких запущенных им потоков, целесообразно использовать функцию ***WaitForMultipleObjects***:

```
DWORD WaitForMultipleObjects (
  DWORD cObjects,          //количество ожидаемых потоков
```

```
  CONST HANDLE *lphObjects,//адрес массива дескрипторов потоков
  BOOL fWaitAll,           //тип ожидания
  DWORD dwTimeout          //время ожидания в мс
);
```

Например, если запущено три потока и их дескрипторы представлены в виде массива `HANDLE hThread[3]`, то ожидание до тех пор, пока все три потока не завершатся, можно организовать следующим образом:

```
WaitForMultipleObjects(3,hThreads,TRUE,INFINITE);
```

Тип ожидания TRUE означает ожидание завершения всех потоков (FALSE – хотя бы одного из потоков). Время ожидания INFINITE означает бесконечное ожидание до наступления требуемого события.

В ОС Windows используется принцип приоритетной диспетчеризации потоков. Это означает, что кванты процессорного времени чаще выделяются потокам с более высоким приоритетом. Значения приоритета устанавливаются в диапазоне от 1 до 31. Существуют 4 уровня приоритетов, которые назначаются процессам при их создании (в зависимости от типа процесса):

– IDLE_PRIORITY_CLASS=4 (низкоприоритетные процессы);
– NORMAL_PRIORITY_CLASS=9 (обычные процессы);
– HIGH_PRIORITY_CLASS=13 (высокоприоритетные процессы);
– REALTIME_PRIORITY_CLASS=24 (процессы реального времени);

Потоки первоначально получают такое же значение приоритета, как и у процесса. Обычные пользовательские процессы (и их потоки) по умолчанию получают значение приоритета 9, что соответствует классу NORMAL_PRIORITY_CLASS. Операционная система может автоматически изменять приоритет потоков в зависимости от их текущего состояния: увеличивать, когда поток взаимодействует с пользователем или снижать, когда поток переходит в состояние ожидания.

С помощью функции ***SetThreadPriority*** можно изменить относительный приоритет потока, но только в рамках установленного класса:

```
BOOL SetThreadPriority (
HANDLE hThread,       //дескриптор потока
int  nPriority        //новый уровень приоритета потока
);
```

Новый уровень приоритета потока задается с помощью специальных констант, которые устанавливают величину изменения приоритета потока относительно приоритета процесса:

– THREAD_PRIORITY_ABOVE_NORMAL (+1)
– THREAD_PRIORITY_HIGHEST (+2)
– THREAD_PRIORITY_NORMAL (0)
– THREAD_PRIORITY_BELOW_NORMAL (–1)
– THREAD_PRIORITY_LOWEST (–2)

– THREAD_PRIORITY_TIME_CRITICAL (=15 (или =31))

Последняя из указанных констант THREAD_PRIORITY_TIME_CRITICAL позволяет установить абсолютное значение приоритета потока, равное 31 для процессов класса REALTIME_PRIORITY_CLASS или 15 для остальных классов.

В любой момент времени можно определить текущее значение приоритета потока с дескриптором hThread с помощью функции ***GetThreadPriority***:

```
int GetThreadPriority (
  HANDLE hThread           // handle потока
);
```

Приведем простой пример работы с потоками в ОС Windows (см. листинг 5.1).

Дана последовательность натуральных чисел, хранящихся в массиве a_0, …, a_{99}. Создадим многопоточное приложение для поиска суммы квадратов элементов этого вектора. Разобьем последовательность чисел на четыре части и создадим четыре потока, каждый из которых будет вычислять суммы квадратов элементов в отдельной части последовательности. Главный поток создаст дочерние потоки, соберет данные и вычислит окончательный результат, после того, как отработают четыре дочерних потока.

Листинг 5.1. Пример работы с потоками с помощью функций WinAPI:

```
#include <stdio.h>
#include <conio.h>
#include <windows.h>
const int n = 4;
int a[100];
DWORD WINAPI ThreadFunc(PVOID pvParam)
{
  int num,sum = 0,i;
  num = 25*(*((int *)pvParam));
  for(i=num;i<num+25;i++)
  {
  sum += a[i]*a[i];
  *(int*)pvParam = sum;
  DWORD dwResult = 0;
  return dwResult;
}
int main(int argc, char** argv)
{
  int x[n];
  int i,rez = 0;
  DWORD dwThreadId[n],dw;
  HANDLE hThread[n];
  for (i=0;i<100;i++)
   a[i] = i;
  //создание n дочерних потоков
  for (i=0;i<n;i++)
```

```
  {
   x[i] = i;
   hThread[i] = CreateThread(NULL,0,ThreadFunc,(PVOID)&x[i], 0,
&dwThreadId[i]);
   if(!hThread)
   printf("main process: thread %d not execute!",i);
  }
  // ожидание завершения n потоков
  dw = WaitForMultipleObjects(n,hThread,TRUE,INFINITE);
  for(i=0;i<n;i++)
   rez+=x[i];
  printf("\nСумма квадратов = %d",rez);
  getch();
  return 0;
}
```

5.4. Потоки в Linux

В Linux API для управления потоками, их синхронизации и планирования определяет стандарт POSIX.1c, Threads extensions (IEEE Std. 1003.1c-1995). Библиотеки, реализующие этот стандарт, называются Pthreads, а функции имеют приставку «pthread_».

В Linux каждый поток на самом деле является процессом, и для того, чтобы создать новый поток, нужно создать новый процесс. В многопоточных приложениях Linux для создания дополнительных потоков используются процессы особого типа. Эти процессы (lightweight processes) представляют собой обычные дочерние процессы главного процесса, но они разделяют с главным процессом адресное пространство, файловые дескрипторы и обработчики сигналов. Прилагательное «легкий» в названии процессов-потоков вполне оправдано. Поскольку этим процессам не нужно создавать собственную копию адресного пространства (и других ресурсов) своего процесса-родителя, создание нового легкого процесса требует значительно меньших затрат, чем создание полновесного дочернего процесса.

Напомним, что в Linux у каждого процесса есть идентификатор. Есть он и у процессов-потоков. Однако спецификация POSIX 1003.1c требует, чтобы все потоки многопоточного приложения имели один идентификатор. Вызвано это требование тем, что для многих функций системы многопоточное приложение должно представляться как один процесс с одним идентификатором. Для решения проблемы единого идентификатора процессы многопоточного приложения группируются в группы потоков (thread groups). Группе присваивается идентификатор, соответствующий идентификатору первого процесса многопоточного приложения. Функция ***getpid()***, возвращает значение идентификатора группы потока, независимо от того, из какого потока она вызвана. Функции ***kill() waitpid()*** и им подобные по умолчанию также используют идентификаторы групп потоков, а не отдельных процессов.

Потоки создаются функцией ***pthread_create()***, определенной в заголовочном файле pthread.h:

```
#include <pthread.h>
int pthread_create
(
  pthread_t * thread, // указатель на идентификатор создаваемого
потока
  pthread_attr_t *attr, // указатель на атрибуты потока
  void *(*start_routine)(void *), // адрес функции потока
  void *arg // значение, возвращаемого функцией потока
);
```

Первый параметр этой функции представляет собой указатель на переменную типа pthread_t, которая служит идентификатором создаваемого потока. Второй параметр, указатель на переменную типа pthread_attr_t, используется для передачи атрибутов потока. Третьим параметром функции pthread_create() должен быть адрес функции потока. Эта функция играет для потока ту же роль, что функция main() для главной программы. Четвертый параметр функции pthread_create() имеет тип void *. Этот параметр может использоваться для передачи значения, возвращаемого функцией потока. Вскоре после вызова pthread_create() функция потока будет запущена на выполнение параллельно с другими потоками программы. Следует учитывать, что перед тем как запустить новую функцию потока, нужно выполнить некоторые подготовительные действия, а поток-родитель между тем продолжает выполняться – это занимает некоторое время. Если в ходе создания потока возникла ошибка, функция pthread_create() возвращает ненулевое значение, соответствующее номеру ошибки.

Функция потока должна иметь заголовок вида:

```
void * func_name(void * arg)
```

Аргумент arg - это указатель, который передается в последнем параметре функции pthread_create(). Функция потока может вернуть значение, которое затем будет проанализировано заинтересованным потоком, но это не обязательно. Завершение функции потока происходит если:

1. Функция потока вызвала функцию pthread_exit().
2. Функция потока достигла точки выхода.
3. Поток был досрочно завершен другим потоком.

Функция ***pthread_exit()*** представляет собой потоковый аналог функции ***_exit()*** и определена следующим образом:

```
# include <pthread.h>
void pthread_exit(
  void *value // значение
);
```

Аргумент value является указателем на данные, возвращаемые потоком, этот указатель может быть получен родительским потоком при помощи функции pthread_join(). Реально при вызове этой функции поток из нее просто не воз-

вращается. Стоит помнить, что функция ***exit()*** по-прежнему завершает процесс, то есть, в том числе уничтожает все потоки.

Для того, чтобы получить значение, возвращенное функцией потока, нужно воспользоваться функцией ***pthread_join()***:

```
# include <pthread.h>
int pthread_join(
  pthread_t threadid, // идентификатор потока
  void *value         // значение
);
```

У этой функции два параметра. Первый параметр – это идентификатор потока, второй параметр имеет тип указатель на нетипизированный указатель. В этом параметре функция возвращает значение, возвращенное функцией потока – таким образом можно организовать передачу данных между потоками. Однако основная задача функции pthread_join() заключается в синхронизации потоков. Функция pthread_join() переводит поток, из которого она была вызвана, в состояние ожидания до тех пор, пока не завершится поток, определяемый идентификатором, переданным в качестве аргумента. Если в момент вызова pthread_join() ожидаемый поток уже завершился, функция вернет управление немедленно. Функцию pthread_join() можно рассматривать как эквивалент ***waitpid()*** для потоков. Попытка выполнить более одного вызова pthread_join() из разных потоков для одного и того же потока приведет к ошибке. При успешном завершении функция возвращает 0. В случае ошибки возвращается ненулевое значение.

Рассмотрим пример программы, реализующей все вышеописанное (см. листинг 5.2). Программа создает процесс и потоки, которые печатают идентификатор процесса и потока.

Листинг 5.2. Пример использования функций pthread_create() и pthread_join:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
void put_msg( char *title, struct timeval *tv ) {
   printf( "%02u:%06lu : %s\t: pid=%lu, tid=%lu\n",
           ( tv->tv_sec % 60 ), tv->tv_usec, title, getpid(),
           pthread_self() );
}
void *test( void *in ) {
   struct timeval *tv = (struct timeval*)in;
   gettimeofday( tv, NULL );
   put_msg( "pthread started", tv );
   sleep( 5 );
   gettimeofday( tv, NULL );
   put_msg( "pthread finished", tv );
   return NULL;
}
```

```
#define TCNT 5
static pthread_t tid[ TCNT ];
int main( int argc, char **argv, char **envp ) {
   pthread_t tid[ TCNT ];
   struct timeval tm;.
   int i;.
   gettimeofday( &tm, NULL );
   put_msg( "main started", &tm );
   for( i = 0; i < TCNT; i++ ) {
      int status = pthread_create( &tid[ i ], NULL, test,
                                   void*)&tm );
      if( status != 0 ) {
         perror( "pthread_create" );
         exit( EXIT_FAILURE );
       }
   };
   for( i = 0; i < TCNT; i++ )
      pthread_join( tid[ i ], NULL ); // ожидание
   gettimeofday( &tm, NULL );
   put_msg( "main finished", &tm );
   return( EXIT_SUCCESS );
}
```

При компиляции надо знать, что хоть функции работы с потоками и описаны в файле включения pthread.h, на самом деле они находятся в библиотеке. Библиотеку libgcc.a рекомендуется скопировать в текущий каталог. В строку компиляции нужно дописать ключ «-lpthread». Откпомилируем пример и выполним его:

```
[root@srv ~]# gcc pthreadexample.c -lpthread -o pthreadexample
[root@srv ~]# ./pthreadexample
50:259188 : main started : pid=14333, tid=3079214784
50:259362 : pthread started   : pid=14333, tid=3079211888
50:259395 : pthread started   : pid=14333, tid=3068722032
50:259403 : pthread started   : pid=14333, tid=3058232176
50:259453 : pthread started   : pid=14333, tid=3047742320
50:259466 : pthread started   : pid=14333, tid=3037252464
55:259501 : pthread finished  : pid=14333, tid=3079211888
55:259501 : pthread finished  : pid=14333, tid=3068722032
55:259525 : pthread finished  : pid=14333, tid=3058232176
55:259532 : pthread finished  : pid=14333, tid=3047742320
55:259936 : main finished     : pid=14333, tid=3079214784
```

В случае если нас чем-то не устраивает возврат значения через pthread_join(), например, нам необходимо получить данные в нескольких нитях, то следует воспользоваться каким либо другим механизмом, например, можно организовать очередь возвращаемых значений, или возвращать значение в структуре, указатель на которую передают в качестве параметра нити. То есть

использование pthread_join() это вопрос удобства, а не догма, в отличие от случая пары fork() и wait() для процессов.

Если предполагается использовать другой механизм возврата или возвращаемое значение не важно - можно отсоединить поток (detach), дав понять ОС, что необходимо освободить ресурсы, связанные с потоком сразу по завершению функции потока. Сделать это можно несколькими способами. Во-первых, можно сразу создать поток отсоединенным, задав соответствующий объект атрибутов при вызове pthread_create(). Во-вторых, любой поток можно отсоединить, вызвав в любой момент его жизни функцию ***pthread_detach()***:

```
#include <pthread.h>
int pthread_detach(
  pthread_t thread // идентификатор нити
);
```

Функция имеет один параметр - идентификатор потока. При этом поток может отсоединить сам себя, получив свой идентификатор при помощи функции ***pthread_self()***:

```
#include <pthread.h>
pthread_t pthread_self(void);
```

Следует подчеркнуть, что отсоединение потока никоим образом не влияет на процесс его выполнения, а просто помечает поток как готовый по своему завершению к освобождению ресурсов: стека, памяти, в которую сохраняется контекст потока, данные специфичные для потока и проч. Сюда не входят ресурсы выделяемые явно, например, память, выделяемая через malloc(), или открываемые файлы. Подобные ресурсы должен отслеживать программист и явно освобождать их сам.

Сделать поток «отдельным» можно и на этапе его создания, с помощью дополнительного атрибута ***DETACHED***. Для того чтобы назначить потоку дополнительные атрибуты, нужно сначала создать объект, содержащий набор атрибутов. Этот объект создается функцией ***pthread_attr_init()***:

```
#include <pthread.h>
int pthread_attr_init(
  pthread_attr_t *attr     // указатель на набор аргументов
);
```

Единственный аргумент этой функции – указатель на переменную типа pthread_attr_t, которая служит идентификатором набора атрибутов. Функция pthread_attr_init() инициализирует набор атрибутов потока значениями, заданными по умолчанию. Для добавления атрибутов в набор используются специальные функции с именами ***pthread_attr_set<имя_атрибута>***. Например, для того, чтобы добавить атрибут «отделенности» используется функция ***pthread_attr_setdetachstate()***:

```
#include <pthread.h>
int pthread_attr_setdetachstate(
  pthread_attr_t *attr,
  int detachstate
);
```

Первым аргументом этой функции должен быть адрес объекта набора атрибутов, а вторым аргументом – константа, определяющая значение атрибута. Константа PTHREAD_CREATE_DETACHED указывает, что создаваемый поток должен быть отделенным, а константа PTHREAD_CREATE_JOINABLE определяет создание присоединяемого (joinable) потока, который может быть синхронизирован функций pthread_join(). После добавления необходимых значений в набор атрибутов потока, необходимо вызвать функцию pthread_create() и передать набор атрибутов потока вторым аргументом.

Точно так же, как при управлении процессами, иногда возникает необходимость досрочно завершить процесс, многопоточной программе может понадобиться досрочно завершить один из потоков. Для досрочного завершения потока можно воспользоваться функцией ***pthread_cancel()***:

```
# include <pthread.h>
int pthread_cancel(
  pthread_t threaded  // идентификатор потока
);
```

Единственным аргументом этой функции является идентификатор потока. Функция pthread_cancel() возвращает 0 в случае успеха и ненулевое значение в случае ошибки. Несмотря на то, что pthread_cancel() может завершить поток досрочно, ее нельзя назвать средством принудительного завершения потоков. Дело в том, что поток может не только самостоятельно выбрать порядок завершения в ответ на вызов pthread_cancel(), но и вовсе игнорировать этот вызов. Вызов функции pthread_cancel() следует рассматривать как запрос на выполнение досрочного завершения потока.

Функция ***pthread_setcancelstate()*** определяет, будет ли поток реагировать на обращение к нему с помощью pthread_cancel(), или не будет.

```
#include <pthread.h>
int pthread_setcancelstate(
  int state,          // новое значение
  int *oldstate  // указатель на старое значение
);
```

Функция pthread_setcancelstate() имеет два параметра - параметр state типа int и параметр oldstate типа «указатель на int». В первом параметре передается новое значение, указывающее, как поток должен реагировать на запрос pthread_cancel(), а в переменную, адрес которой был передан во втором параметре, функция записывает прежнее значение. State может иметь два значения PTHREAD_CANCEL_DISABLE (запретить досрочное завершение потока) и

PTHREAD_CANCEL_ENABLE (разрешить досрочное завершение потока). Если прежнее значение не интересует, во втором параметре можно передать NULL. Функция возвращает 0 в случае успеха и ненулевое значение в случае ошибки.

Чаще всего функция pthread_setcancelstate() используется для временного запрета завершения потока путем ограждения фрагмента кода, во время выполнение которого завершать поток крайне нежелательно:

```
pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, NULL);
... //Здесь поток завершать нельзя
pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);
```

Если запрос на досрочное завершение потока поступит в тот момент, когда поток игнорирует эти запросы, выполнение запроса будет отложено до тех пор, пока функция pthread_setcancelstate() не будет вызвана с разрешающим аргументом. Что именно произойдет дальше, зависит от более тонких настроек потока.

Если досрочное завершение разрешено, поток, получивший запрос на досрочное завершение, может завершить работу не сразу. Если поток находится в режиме отложенного досрочного завершения (именно этот режим установлен по умолчанию), он выполнит запрос на досрочное завершение, только достигнув одной из точек отмены. В соответствии со стандартом POSIX, точками отмены являются вызовы многих обычных функций, например open(), pause() и write(). Установить точку отмены вручную можно с помощью функции ***pthread_testcancel()***:

```
#include <pthread.h>
void pthread_testcancel(void);
```

В частности, установить явную точку отмены может потребоваться при использовании функции printf(), т.к. при её вызове поток завершается, но pthread_join() не возвращает управление.

Впрочем, можно выполнить досрочное завершение потока, не дожидаясь точек останова. Для этого необходимо перевести поток в режим немедленного завершения, что делается с помощью вызова

```
pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS, NULL);
```

В этом случае беспокоиться о точках останова уже не нужно. Вызов

```
pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED, NULL);
```

снова переводит поток в режим отложенного досрочного завершения.
Рассмотрим пример программы (см. листинг 5.3).

Листинг 5.3. Пример использования функции pthread_setcancelstate():

```
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>
```

```
int i = 0;
void * thread_func(void *arg)
{
  pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, NULL);
  for (i=0; i < 4; i++)
  {
    sleep(1);
    printf("I'm still running!\n");
  }
  pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);
  pthread_testcancel();
  printf("YOU WILL NOT STOP ME!!!\n");
}
int main(int argc, char * argv[])
{
  pthread_t thread;
  pthread_create(&thread, NULL, thread_func, NULL);
  while (i < 1) sleep(1);
  pthread_cancel(thread);
  printf("Requested to cancel the thread\n");
  pthread_join(thread, NULL);
  printf("The thread is stopped.\n");
  return EXIT_SUCCESS;
}
```

В самом начале функции потока thread_func() мы запрещаем досрочное завершение потока, затем выводим четыре тестовых сообщения с интервалом в одну секунду, после чего разрешаем досрочное завершение. Далее, с помощью функции pthread_testcancel(), создаем точку отмены (cancellation point) потока. Если досрочное завершение потока было затребовано, в этот момент поток должен завершиться. Затем мы выводим еще одно диагностическое сообщение, которое пользователь не должен видеть, если программа сработает правильно.

В главной функции программы мы создаем поток, затем дожидаемся, пока значение глобальной переменной i станет больше нуля (это гарантирует нам, что поток уже запретил досрочное завершение) и вызываем функцию pthread_cancel(). После этого мы переходим к ожиданию завершения потока с помощью pthread_join(). Если вы скомпилируете и запустите программу, то увидите, что поток распечатает четыре тестовых сообщения I’m still running! (после первого сообщения главная функция программы выдаст запрос на завершение потока).

Поскольку поток завершится досрочно, последнего тестового сообщения вы не увидите.

Предположим, что поток выделяет блок динамической памяти и затем внезапно завершается по требованию другого потока. Если бы поток был самостоятельным процессом, ничего особенно неприятного не случилось бы, так как система сама убрала бы за ним мусор. В случае же процесса-потока блок останется невысвобожденым, что в конечном итоге приведет к серьезным утечкам памяти.

Таким образом, для эффективного управления завершением потоков необходим еще и механизм, оповещающий поток о досрочном завершении. Если нужно выполнять какие-то специальные действия в момент завершения потока (нормального или досрочного), мы устанавливаем функцию-обработчик, которая будет вызвана перед тем, как поток завершит свою работу.

Для установки обработчика завершения потока применяется макрос ***pthread_cleanup_push()***:

```
#include <pthread.h>
void pthread_cleanup_push(
  void (*routine)(void *), // адрес функции-обработчика
  void *arg                // аргументы
);
```

У макроса два аргумента. В первом аргументе ему должен быть передан адрес функции-обработчика завершения потока, а во втором – нетипизированный указатель, который будет передан как аргумент при вызове функции-обработчика. Этот указатель может указывать на что угодно, мы сами решаем, какие данные должны быть переданы обработчику завершения потока. Макрос pthread_cleanup_push() помещает переданные ему адрес функции-обработчика и указатель в специальный стек. Потоку можно назначить произвольное число функций-обработчиков завершения. Поскольку в стек записывается не только адрес функции, но и ее аргумент, мы можем назначить один и тот же обработчик с несколькими разными аргументами.

Извлечение обработчиков из стека и их выполнение может производиться либо явно, либо автоматически. Автоматически обработчики завершения потока выполняются при вызове потоком функции pthread_exit(), завершающей работу потока, а также при выполнении потоком запроса на досрочное завершение. Явным образом обработчики завершения потока извлекаются из стека с помощью макроса ***pthread_cleanup_pop()***:

```
#include <pthread.h>
void pthread_cleanup_pop(
  int execute
);
```

Аргумент макроса pthread_cleanup_pop() позволяет указать, следует ли выполнять функцию-обработчик, или требуется только удалить ее из стека.

Следует помнить, что макрос pthread_cleanup_pop() должен быть вызван столько же раз, сколько и макрос pthread_cleanup_push().

Рассмотрим методы назначения и выполнения обработчиков завершения потока на простом примере (см. листинг 5.4):

Листинг 5.4. Пример использования обработчиков завершения потоков:

```
#include <stdlib.h>
#include <stdio.h>
```

```
#include <errno.h>
#include <pthread.h>

void exit_func(void * arg)
{
  free(arg);
  printf("Freed the allocated memory.\n");
}

void * thread_func(void *arg)
{
  int i;
  void * mem;
  pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, NULL);
  mem = malloc(1024);
  printf("Allocated some memory.\n");
  pthread_cleanup_push(exit_func, mem);
  pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);
  for (i = 0; i < 4; i++)
  {
    sleep(1);
    printf("I'm still running!!!\n");
  }
  pthread_cleanup_pop(1);
}

int main(int argc, char * argv[])
{
  pthread_t thread;
  pthread_create(&thread, NULL, thread_func, NULL);
  pthread_cancel(thread);
  pthread_join(thread, NULL);
  printf("Done.\n");
  return EXIT_SUCCESS;
}
```

Поток начинает работу с того, что запрещает свое досрочное завершение. Далее поток выделяет блок памяти. Для того чтобы избежать утечек памяти используется функция-обработчик завершения потока exit_func(), которая добавляется в стек обработчиков завершения потока с помощью макроса pthread_cleanup_push(). Вторым параметром функции устанавливается указатель на блок памяти. Функция exit_func() высвобождает блок памяти с помощью функции ***free()*** и выводит диагностическое сообщение. После установки обработчика завершения потока поток разрешает досрочное завершение. Однако если бы поток завершился после выделения блока памяти, но до назначения функции-обработчика, выделенный блок не был бы удален. Перед выходом из функции потока вызывается макрос pthread_cleanup_pop().

Задание

1. Изучить краткие теоретические сведения и лекционный материал по теме практического задания.
2. Реализовать приведенные примеры программы и продемонстрировать их работу.
3. Получить индивидуальное задание у преподавателя. Выбрать модель многопоточного приложения, наиболее точно отвечающую специфике задачи. Разработать алгоритм решения задания, с учетом разделения вычислений между несколькими потоками. Желательно избегать ситуаций изменения одних и тех же общих данных несколькими потоками. Если же избежать этого невозможно, необходимо использовать алгоритмы с активным ожиданием или неделимые операции.
4. Реализовать алгоритм с применением функций библиотеки Pthread.
5. Реализовать алгоритм с применением функций WinAPI.
6. Сравнить возможности обоих подходов, сделать выводы.

Контрольные вопросы

1. Дайте определения понятиям «процесс» и «поток». Чем они отличаются и какие сходства имеют?
2. Какие модели построения многопоточных приложений вам известны?
3. Что такое главный, родительский и дочерний потоки?
4. Поясните параметры функции CreateThread.
5. Что такое функция потока и как передаются параметры функции потока?
6. Что такое дескриптор потока и как он используется при управлении потоком?
7. Перечислите и поясните способы завершения выполнения потока.
8. Как формируются и регулируются приоритеты процессов и потоков?
9. Какими способами родительский поток может получить информацию о текущем состоянии дочернего потока (завершен или еще выполняется)?
10. С какой целью при создании многопоточных приложений используется функция WaitForMultipleObject и каковы ее параметры?
11. Что такое многопоточность, чем многопоточность отличается от многозадачности?
12. В чем отличие потоков в в ОС семейства Unix/Linux от потоков в ОС Windows? Что такое «облегченный процесс»?
13. Перечислите и поясните основные модели создания многопоточного приложения?

14. Каким образом и с помощью чего создаются потоки в операционных системах семейства Unix/Linux? Поясните параметры функции pthread_create.

15. Каким образом в ОС семейства Unix/Linux идентифицируются потоки одного процесса? Что такое группы потоков?

16. В каких случаях происходит завершение функции потока в ОС семейства Unix/Linux?

17. Как получить значение, возвращенное функцией потока в ОС семейства Unix/Linux?

18. Что такое «отсоединенный поток»? Как можно «отсоединить» поток?

19. Каким образом можно досрочно завершить поток в ОС семейства Unix/Linux?

20. Что произойдет, если запрос на досрочное завершение потока поступит в тот момент, когда поток игнорирует эти запросы?

21. Что такое «точка отмены» и как её можно установить?

22. Каким образом можно оповестить поток о досрочном завершении? Каким образом работает обработчик завершения потока?

Варианты заданий

1. Для Windows и Linux написать программы, разбивающие заданный интервал на диапазоны и осуществляющую поиск простых чисел в каждом из интервалов с помощью потоков.

2. Для Windows и Linux написать программы, разбивающие входной файл на фрагменты и осуществляющие поиск подстроки в каждом из фрагментов с помощью потоков.

3. Для Windows и Linux написать программы, умножающие матрицу произвольного порядка на вектор, при этом умножение каждой строки на вектор производить в отдельном потоке.

4. Для Windows и Linux написать программы, осуществляющие проверку доступности диапазона IP-адресов сети класса «С» (254 адреса, маска 255.255.255.0), разделенного на несколько поддиапазонов. Проверку доступности узла сети на доступность реализовать в отдельном потоке.

5. Координаты заданного количества шариков изменяются на случайную величину по вертикали и горизонтали, при выпадении шарика за нижнюю границу допустимой области шарик исчезает. Напишите программу (для Windows и Linux) изменения координат заданного количества шариков, где изменение координат каждого шарика осуществляется функцией в отдельном потоке.

6. Противостояние двух команд – каждая команда увеличивается на случайное количество бойцов и убивает случайное количество бойцов участника. Разработать программу (для Windows и Linux), вызывающую из

главной функции в отдельном потоке для каждой команды функцию, изменяющую число бойцов (уменьшение числа бойцов в противостоящей команде и увеличение в своей на случайную величину).

7. Для Windows и Linux написать программы, копирующие содержимое директории пофайлово, при этом главная функция программы должна вызывать отдельные потоки для копирования каждой из групп файлов.

8. Для Windows и Linux написать программы, вычисляющие контрольную сумму всех файлов в директории, при этом обработка каждого отдельного файла осуществляется в отдельном потоке.

9. Для Windows и Linux написать программы, вычисляющие математическое ожидание и дисперсию в нескольких массивах данных, при этом обработка каждого массива осуществляется в отдельном потоке.

10. Для Windows и Linux написать программы, вычисляющие частоты биграмм символов (аа, аб, ав, … яэ, яю, яя) в тексте. При этом программа должна разбивать текст на отдельные фрагменты и осуществлять обработку каждого фрагмента в отдельном потоке.

11. Для Windows и Linux написать программы, вычисляющие частоты триграмм символов (ааа, ааб, аав, … яяэ, яяю, яяя) в тексте. При этом программа должна разбивать текст на отдельные фрагменты и осуществлять обработку каждого фрагмента в отдельном потоке.

12. Для Windows и Linux написать программы, вычисляющие частоты биграмм в тексте. Например, для предыдущего предложения биграммы слов это пары «Для Windows», «Windows и», «и Linux», «Linux написать» и т.д. При этом программа должна разбивать текст на отдельные фрагменты и осуществлять обработку каждого фрагмента в отдельном потоке.

13. Для Windows и Linux написать программы, вычисляющие частоты триграмм в тексте. Например, для предыдущего предложения биграммы слов это пары «Для Windows и», «Windows и Linux», «и Linux написать», «Linux написать программы» и т.д. При этом программа должна разбивать текст на отдельные фрагменты и осуществлять обработку каждого фрагмента в отдельном потоке.

14. Медведь и пчелы. Заданное количество пчел добывают мед равными порциями, задерживаясь в пути на случайное время. Медведь потребляет мед порциями заданной величины за заданное время и столько же времени может прожить без питания. Для Windows и Linux написать программы, осуществляющие работу медведя, при этому для заданного количества пчел вызывается отдельный поток, реализующий работу одной пчелы.

15. Авиаразведка - создается условная карта в виде матрицы, размерность которой определяет размер карты, содержащей произвольное количество единиц (целей) в произвольных ячейках. Из произвольной точки карты стартуют несколько разведчиков (потоков), курсы которых выбираются так, чтобы покрыть максимальную площадь карты. Каждый разведчик фикси-

рует цели, чьи координаты совпадают с его координатами и по достижении границ карты сообщает количество обнаруженных целей. Реализуйте соответствующие программы для Windows и Linux, используя потоки.

16. Бег с препятствиями - создается условная карта трассы в виде матрицы, ширина которой соответствует количеству бегунов, а высота – фиксирована, содержащей произвольное количество единиц (препятствий) в произвольных ячейках. Стартующие бегуны (потоки) перемещаются по трассе и при встрече с препятствием задерживаются на фиксированное время. По достижении финиша бегуны сообщают свой номер. Реализуйте соответствующие программы для Windows и Linux, используя потоки.

Тема № 6

Синхронизация потоков и процессов

Цель работы

Изучить средства синхронизации потоков и процессов. Познакомиться с соответствующими функциями WinAPI и POSIX API. В процессе изучения основного материала познакомиться также с функциями отображения файлов на память и директивами препроцессора.

Краткие теоретические сведения

6.1. Синхронизация, общие сведения

В случае если два или более процесса или потока используются один общий разделяемый ресурс, возникает проблема синхронизации. Например, несколько процессов обрабатывают данные из одного файла, используют общую переменную и т.д. Часть программы, в которой осуществляется доступ к разделяемым данным, называется **критической секцией**. Проблема может решаться приостановкой и активизацией процессов, организацией очередей, блокированием и освобождением ресурсов. Пренебрежение вопросами синхронизации может привести к неправильной работе программы, порче данных и критическим ошибкам операционной системы из-за возникновения следующих ситуаций.

1. **Взаимоблокировки (тупики)**. Процессы находится в тупиковой ситуации, если каждый процесс из группы ожидает события, которое может вызвать только другой процесс из этой же группы.

2. **Голодание (бесконечная отсрочка)**. Остановка работы одного или нескольких процессов на неопределенное время вследствие исполнения процессов с большим или равным приоритетом и/или длительным временем пробуждения процесса или разблокировки.

3. **Гонка данных** - ситуация, при которой конечное состояние системы зависит от порядка и интенсивности выполнения потоков, когда потоки не имеют информации друг о друге, но работают с общим ресурсом и хотя бы один из них изменяет этот ресурс.

Для синхронизации потоков, принадлежащих разным процессам, ОС должна предоставлять потокам системные объекты синхронизации. Рассмотрим основные.

1. **Условные переменные** - блокирование одного или нескольких потоков до момента поступления сигнала от другого потока о выполнении некоторого условия или до истечения максимального промежутка времени ожидания. Над переменной можно выполнять две основные операции: «ожидание» и «сигнал». Поток, выполнившая операцию «ожидание», блокируется до того момента, пока другая нить не выполнит операцию «сигнал». Таким образом, операцией «ожидание» первый поток сообщает системе, что он ждет выполнения ка-

кого-то условия, а операцией «сигнал» второй поток сообщает первой, что параметры, от которых зависит выполнение условия, возможно, изменились. Основное применение условных переменных – это сценарий «производитель-потребитель».

Рассмотрим две нити, одна из которых генерирует данные, а другая – перерабатывает их. В простейшем случае производитель помещает каждую следующую порцию данных в разделяемую переменную, а потребитель считывает ее оттуда. При этом могут возникать две проблемы. Если производитель работает быстрее потребителя, то он может записать очередную порцию данных до того, как потребитель прочитает предыдущую. При этом предыдущая порция данных будет потеряна. Если же потребитель работает быстрее производителя, он может обработать одну и ту же порцию данных несколько раз. Классическое решение этой задачи реализуется с использованием условной переменной.

2. **Блокировки чтения-записи** можно использовать для реализации стратегии доступа «параллельное чтение и исключающая запись». Имеют два режима захвата: для чтения и для записи. Блокировку для чтения могут удерживать несколько потоков одновременно. Блокировку для записи может удерживать только один поток. При этом никакой другой поток не может удерживать эту же блокировку для чтения. Используются блокировки для защиты структур данных, которые читают значительно чаще, чем модифицируют.

3. **Барьеры** – используются для синхронизации потоков, выполняющих части одной и той же работы, например, параллельные вычисления и позволяют гарантировать, что даже при неравномерной загрузке системы, потоки будут выполнять равный объем работы в единицу времени. Пусть N - количество потоков, необходимое для перехода через барьер. Потоки, подходящие к барьеру, вызывают функцию «ожидание». Если количество потоков, ожидающих возле барьера, меньше N-1, поток блокируется. Когда набирается N потоков, все они разблокируются и продолжают исполнение. Барьеры полезны для организации коллективных распределенных вычислений в многопроцессорной конфигурации, когда каждый участник (поток управления) выполняет часть работы, а в точке сбора частичные результаты объединяются в общий итог.

4. **Семафор** – представляет собой целочисленную переменную S, над которой определены две операции $P(S)$ и $V(S)$. $V(S)$ – переменная S увеличивается на 1 атомарным действием (выборка, наращивание и запоминание не могут быть прерваны). К переменной S нет доступа во время выполнения этой операции. $P(S)$ – переменная S уменьшается на 1 атомарным действием, если это возможно, оставаясь при этом в области неотрицательных значений. Если S уменьшить невозможно, поток, выполняющий операцию P, ждет, пока это уменьшение станет возможным. Операция P заключает в себе потенциальную возможность перехода процесса, который ее выполняет, в состояние ожидания (если $S = 0$). Операция V может при некоторых обстоятельствах активизировать процесс, приостановленный операцией P. Иногда семафоры используют в качестве разделяемых целочисленных переменных, например в качестве счетчиков записей в очереди. Использование семафоров обеспечит безопасное использование ресурса только в том случае, если все потоки управления будут захватывать семафор перед ис-

пользованием ресурса, и освобождать его, как только необходимость в нем отпадет. За взаимосвязь ресурсов и семафоров отвечает прикладная программа. Семафоры также можно использовать для обхода проблемы инверсии приоритета. Если высокоприоритетный и низкоприоритетный потоки должны взаимодействовать, иногда удается реализовать соглашение, что только низкоприоритетный поток может выполнять над семафором операцию уменьшения семафора, а высокоприоритетный поток может делать только операцию увеличения семафора. Такое взаимодействие обычно похоже на схему «производитель-потребитель» с тем отличием, что производитель может терять некоторые порции данных, если потребитель за ним не успевает.

5. **Мьютексы** – представляют собой двоичный семафор. Могут находиться в одном из двух состояний: отмеченном или неотмеченном. Когда поток становится владельцем мьютекса, последний переводится в неотмеченное состояние. Если задача освобождает мьютекс, его состояние становится отмеченным. Мьютексы используются для управления критическими разделами потоков, чтобы предотвратить возникновение условий гонки данных за счет реализации последовательного доступа к критическим секциям.

6. **Критическая секция** – позволяет предотвратить одновременное выполнение некоторого набора операций (обычно связанных с доступом к данным) несколькими потоками. Может использоваться только в пределах одного потока. Выполняет те же задачи, что и мьютекс. Процедура, аналогичная захвату мьютекса, называется входом в критическую секцию. Снятие блокировки мьютекса называется выходом из критической секции. Выполнение обеих операций занимает меньшее время, чем аналогичные операции мьютекса, что связано с отсутствием необходимости обращаться к ядру ОС.

7. **Ожидающие таймеры** – объекты ядра, которые предназначены для отсчета промежутков времени. Они самостоятельно переходят в свободное состояние в определенное время или через регулярные промежутки времени. Применяется для периодического выполнения задачи. Окончание временного интервала определяется по переходу таймера в свободное состояние. Момент перехода таймера в свободное состояние определяется одной из ожидающих функций.

8. **Спин-блокировки** – низкоуровневое средство синхронизации, предназначенное в для применения в многопроцессорной конфигурации с разделяемой памятью. Аналогично мьютексам могут иметь два значения и реализуются как атомарно устанавливаемое булево значение: истина – блокировка установлена, ложь – блокировка снята. Заметим, что все ожидания в спин-блокировках непрерываемые. При попытке установить спин-блокировку, если она захвачена кем-то другим, как правило, применяется активное ожидание освобождения с постоянным опросом в цикле состояния блокировки. Активное ожидание и установка не связаны с переключением контекстов, активизацией планировщика и т.п. Спин-блокировка устанавливается на очень короткий участок кода и должна быть реализована весьма эффективно, чтобы накладные расходы не оказались высокими. Если ожидание оказывается кратким, минимальными оказываются и накладные расходы. В этом заключает основное преимущество спин-блокировок перед мьютексами.

Рассмотрим, каким образом данные синхронизирующие объекты реализованы в операционных системах.

6.2. Средства синхронизации потоков в Windows

В ОС Windows существуют следующие средства синхронизации:
1) семафоры (Semaphore);
2) мьютексы (Mutex);
3) события (Event);
4) критические секции (Critical Section) и спин-блокировки;
5) таймеры ожидания (Waitable Timer).

6.2.1 Семафоры

Основные функции для работы с семафорами это создание семафора ***CreateSemaphore()*** и увеличение счетчика семафора ***ReleaseSemaphore()***:

```
HANDLE CreateSemaphore // создание семафора
// Возвращает идентификатор семафора, иначе NULL
(
  LPSECURITY_ATTRIBUTES lpSemaphoreAttributes,  // атрибут доступа
  LONG lInitialCount, // начальное состояние счетчика
  LONG lMaximumCount, // максимальное количество обращений
  LPCTSTR lpName      // имя объекта
);

BOOL ReleaseSemaphore // увеличение счетчика
// При успешном выполнении возвращаемое значение ненулевое
(
 HANDLE hSemaphore,   // хендл семафора
 LONG lReleaseCount,  // значение инкремента (положительное)
 LPLONG lpPreviousCount    // предыдущее значение
);
```

В случае, когда необходимо синхронизовать задачи разных процессов, следует определить имя семафора. При этом один процесс создает семафор с помощью функции CreateSemaphore(), а второй открывает его, получая идентификатор для уже существующего семафора. Открыть существующий семафор можно с помощью функции ***OpenSemaphore()***:

```
HANDLE OpenSemaphore(           // открытие семафора
  DWORD   fdwAccess,            // требуемый доступ
  BOOL    fInherit,             // флаг наследования
  LPCTSTR lpszSemaphoreName     // адрес имени семафора
);
```

В API операционной системы Microsoft Windows нет функции, специально предназначенной для уменьшения значения счетчика семафора. Этот счетчик уменьшается, когда задача вызывает функции ожидания, такие как ***WaitForSingleObject()*** или ***WaitForMultipleObject()***. Если задача вызывает не-

сколько раз функцию ожидания для одного и того же семафора, содержимое его счетчика каждый раз будет уменьшаться.

Удаляется семафор также как и все другие объекты ядра функцией **CloseHandle:**

```
BOOL CloseHandle      // удалить семафор (как объект)
(
 HANDLE hobj          // передать хендл семафора
);
```

Рассмотрим пример использования семафоров (см. листинг 6.1). Запускаются три потока, каждый из которых обращается к семафору. Однако так как значение семафора равно двум, то к нему разрешается обращаться всего двум потокам. Поэтому третий поток будет ждать, пока кто-то освободит семафор. Также обратите внимание на то, что вместо использования CreateThread(), поток создается с помощью функции ***_beginthread(),*** а завершается функцией ***_endthread()*** – это ещё один из возможных способов работы с потоками (стандарт ANSI).

Листинг 6.1. Пример использования семафоров:

```
#include "stdafx.h"
#include "windows.h"
#include "iostream.h"
#include "process.h"

HANDLE hSemaphore;
LONG cMax = 2;

void Test1(void *);
void Test2(void *);
void Test3(void *);

void main()
{
 hSemaphore = CreateSemaphore(
     NULL,  // нет атрибута
     cMax,  // начальное состояние = 2
     cMax,  // максимальное состояние = 2
     NULL   // семафор без имени
 );

 if (!hSemaphore == NULL)
 {
  if (_beginthread(Test1,1024,NULL)==-1)
   cout << "Error begin thread " << endl;
  if (_beginthread(Test2,1024,NULL)==-1)
   cout << "Error begin thread " << endl;
  if (_beginthread(Test3,1024,NULL)==-1)
  cout << "Error begin thread " << endl;
```

```
   Sleep(10000);
   CloseHandle(hSemaphore);
  }
  else
   cout << "error create semaphore" << endl;
}

void Test1(void *)
{
 cout << "Test1 Running" << endl;
 DWORD dwWaitResult;
 while(dwWaitResult!=WAIT_OBJECT_0)
 {
  dwWaitResult = WaitForSingleObject(
   hSemaphore,  // указатель на семафор
   1            // интерфал ожидания
  );
  cout << "Test 1 TIMEOUT" << endl;
 }
 Sleep(1000);
 if (ReleaseSemaphore(
    hSemaphore,  // указатель на светофор
    1,           // изменяет счетчик на 1
    NULL)
 )
 cout << " ReleaseSemaphore Ok Test1" << endl;
 _endthread();
}

void Test2(void *)
{
 cout << "Test2 Running" << endl;
 DWORD dwWaitResult;
 while(dwWaitResult!=WAIT_OBJECT_0)
 {
  dwWaitResult = WaitForSingleObject(hSemaphore,1);
  cout << "Test 2 TIMEOUT" << endl;
 }
 Sleep(1000);
 if (ReleaseSemaphore(hSemaphore,1,NULL))
  cout << " ReleaseSemaphore Ok Test2" << endl;
 _endthread();
}

void Test3(void *)
{
 cout << "Test2 Running" << endl;
 DWORD dwWaitResult;
 while(dwWaitResult!=WAIT_OBJECT_0)
```

```
 {
  dwWaitResult = WaitForSingleObject(hSemaphore,1);
  cout << "Test 3 TIMEOUT" << endl;
 }
 if (ReleaseSemaphore(hSemaphore,1,NULL))
  cout << " ReleaseSemaphore Ok Test3" << endl;
 _endthread();
}
```

6.2.2 Мьютексы

Для создания мьютекса предназначена функция ***CreateMutex()***, для освобождения ***ReleaseMutex()***. Удалить мьютекс можно с помощью функции ***CloseHandle()***.

```
HANDLE CreateMutex // создать мьютекс
// Возвращает дескриптор объекта mutex, а если такое имя есть, то
// дескрпитор существующего.
(
  LPSECURITY_ATTRIBUTES lpMutexAttributes, // атрибут безопасности
  BOOL bInitialOwner,       // флаг начального владельца
  LPCTSTR lpName            // имя объекта
);

BOOL ReleaseMutex           // освободить мьютекс
(
  HANDLE hMutex             // дескриптор mutex
);
```

Листинг 6.2. Пример использования мьютекса:

```
#include "stdafx.h"
#include "windows.h"
#include "iostream.h"
void main()
{
 HANDLE mut;
 mut = CreateMutex(NULL, FALSE, "Mutex"); // создаем мьютекс
 DWORD result;
 result = WaitForSingleObject(mut,0); // если его удается захватить
 if (result == WAIT_OBJECT_0)
 { // захватили, выполняем программу
  cout << "program running" << endl;
  int i;
  cin >> i;
  ReleaseMutex(mut); // освобождаем мьютекс
 }
 else // не удалось захватить
  cout << "fail program running" << endl;
 CloseHandle(mut); // удаляем мьютекс
}
```

6.2.3 События

Объект Событие создается с помощью функции ***CreateEvent()***:

```
HANDLE CreateEvent    // создать объект событие
// В случае успеха вернет дескриптор события. Если событие с таким
// именем уже создано - вернется дескриптор уже созданного события
(
  LPSECURITY_ATTRIBUTES lpEventAttributes, // атрибут защиты
  BOOL bManualReset,        // тип сброса TRUE – ручной
  BOOL bInitialState,       // начальное состояние TRUE - сигнальное
  LPCTSTR lpName            // имя объекта
);
```

Изменить состояние события на сигнальное можно с помощью функции **SetEvent()**:

```
BOOL SetEvent         // изменить состояние на сигнальное
// В случае успеха вернет ненулевое значение
(
 HANDLE hEvent        // дескриптор события
);
```

Изменить состояние события на несигнальное можно с помощью функции **ResetEvent()**:

```
BOOL ResetEvent       // изменить состояние на несигнальное
(
 HANDLE hEvent        // дескриптор события
);
```

В следующем примере создается объект Event, запускается три потока. Каждый поток ждет, когда объект синхронизации перейдет в сигнальное состояние. После некоторой задержки в основной процедуре программы устанавливаем его в сигнальное состояние, выжидаем некоторое время, чтобы потоки среагировали и сбрасываем. Обратите внимание, что в данном случае объектов синхронизации потоков может быть любое количество.

Листинг 6.3. Пример использования событий:

```
#include "stdafx.h"
#include "windows.h"
#include "iostream.h"
#include "process.h"

HANDLE event;

void Test1(void *);
void Test2(void *);
void Test3(void *);
```

```
void main()
{
 event=CreateEvent(NULL,TRUE,FALSE,"FirstStep");
 if (_beginthread(Test1,1024,NULL)==-1)
   cout << "Error begin thread " << endl;
 if (_beginthread(Test2,1024,NULL)==-1)
   cout << "Error begin thread " << endl;
 if (_beginthread(Test3,1024,NULL)==-1)
   cout << "Error begin thread " << endl;
 if (event!=NULL){
   Sleep(1000);
   SetEvent(event);
   Sleep(1000);
   ResetEvent(event);
   CloseHandle(event);
 } else {
   cout << "error create event" << endl;
 }
}

void Test1(void *)
{
 DWORD dwWaitResult;
 while(dwWaitResult!=WAIT_OBJECT_0) {
   dwWaitResult = WaitForSingleObject(event,1);
   cout << "Test 1 TIMEOUT" << endl;
 }
 cout << "Event Test 1 " << endl;
 _endthread();
}

void Test2(void *)
{
 DWORD dwWaitResult;
 while(dwWaitResult!=WAIT_OBJECT_0){
   dwWaitResult = WaitForSingleObject(event,1);
   cout << "Test 2 TIMEOUT" << endl;
 }
 cout << "Event Test 2 " << endl;
 _endthread();
}

void Test3(void *)
{
 DWORD dwWaitResult;
 while(dwWaitResult!=WAIT_OBJECT_0){
   dwWaitResult = WaitForSingleObject(event,1);
   cout << "Test 3 TIMEOUT" << endl;
```

```
 }
 cout << "Event Test 3 " << endl;
 _endthread();
}
```

6.2.4 Критические секции

Для инициализации критической секции используется функция ***InitializeCriticalSection()***:

```
VOID InitializeCriticalSection  // инициализация кр. секции
(
  LPCRITICAL_SECTION lpCriticalSection // указатель на кр. секцию
);
```

Для обозначения точки входа и выхода в/из критической секции применяются соответственно функции ***EnterCriticalSection()*** и ***LeaveCriticalSection()***:

```
VOID EnterCriticalSection  // объявление начала критической секции
(
  LPCRITICAL_SECTION lpCriticalSection  // указатель на кр. секцию
);

VOID LeaveCriticalSection       // выход из критической секции
(
  LPCRITICAL_SECTION lpCriticalSection  // указатель на кр. секцию
);
```

В следующем примере объект array, над которым можно производить некие действия, объявлен глобально и к нему может обратиться любой из потоков. В случае если количество работающих с этим объектом потоков превышает один, может возникнуть ряд проблем: одна функция не успеет очистить массив, а вторая может уже начать писать или печатать не до конца очищенный массив. Тот код, который правит распределенный ресурс, является критической секцией и выделен соответствующими функциями. Благодаря их использованию потоки дожидаются своей очереди и только тогда выполняют необходимые действия.

Листинг 6.4. Пример использования критических секций:

```
#include "stdafx.h"
#include "windows.h"
#include "iostream.h"
#include "process.h"

#define MAX_ARRAY 5

CRITICAL_SECTION critsect;

int array[MAX_ARRAY];
```

```
void EmptyArray(void *);
void PrintArray(void *);
void FullArray(void *);

void main()
{
 InitializeCriticalSection(&critsect);
 if (_beginthread(EmptyArray,1024,NULL)==-1)
  cout << "Error begin thread " << endl;
 if (_beginthread(PrintArray,1024,NULL)==-1)
  cout << "Error begin thread " << endl;
 if (_beginthread(FullArray,1024,NULL)==-1)
  cout << "Error begin thread " << endl;
 if (_beginthread(PrintArray,1024,NULL)==-1)
  cout << "Error begin thread " << endl;
 if (_beginthread(EmptyArray,1024,NULL)==-1)
  cout << "Error begin thread " << endl;
 if (_beginthread(PrintArray,1024,NULL)==-1)
  cout << "Error begin thread " << endl;
 Sleep(10000);
}

void EmptyArray(void *)
{
 cout << "EmptyArray" << endl;
 EnterCriticalSection(&critsect);
 for (int x=0;x<(MAX_ARRAY+1); x++) array[x]=0;
 Sleep(1000);
 LeaveCriticalSection(&critsect);
 _endthread();
}

void PrintArray(void *)
{
 cout << "PrintArray" << endl;
 EnterCriticalSection(&critsect);
 for (int x=0;x<(MAX_ARRAY+1); x++) cout << array[x] << " ";
 cout << endl;
 Sleep(1000);
 LeaveCriticalSection(&critsect);
 _endthread();
}

void FullArray(void *)
{
 cout << "FullArray" <<
 endl;
 EnterCriticalSection(&critsect);
```

```
 for (int x=0;x<(MAX_ARRAY+1); x++) array[x]=x;
 Sleep(1000);
 LeaveCriticalSection(&critsect);
 _endthread();
}
```

Для использования спин-блокировки в критической секции нужно инициализировать счетчик циклов, вызвав функцию ***InitializeCriticalSectionAndSpinCount()***:

```
BOOL WINAPI InitializeCriticalSectionAndSpinCount(
  LPCRITICAL_SECTION lpCriticalSection, // указатель на кр. секцию
  DWORD dwSpinCount                     // счетчик
);
```

6.2.5 Ожидающие таймеры

Для создания таймера применяется функция ***CreateWaitableTimer()***:

```
HANDLE CreateWaitableTimer // создает таймер в занятом состоянии,
// из которого он выводится принудительно (после создания объект не
// активен)
(
  LPSECURITY_ATTRIBUTES lpTimerAttributes, // атрибуты таймера
  BOOL bManualReset, // автосброс (запускает один поток) или ручной
  //(запускаются все потоки, которые его ждали)
  LPCTSTR lpTimerName // имя таймера
);
```

Параметр fManualReset определяет тип ожидаемого таймера: со сбросом вручную или с автосбросом. Когда освобождается таймер со сбросом вручную, возобновляется выполнение всех потоков, ожидавших этот объект, а когда в свободное состояние переходит таймер с автосбросом - лишь одного из потоков.

Объекты данного типа всегда создаются в занятом состоянии. Для запуска и настройки таймера используется функция ***SetWaitableTimer():***

```
BOOL SetWaitableTimer // запускает таймер и настраивает. Может быть
// вызвана в любой момент, но параметры применятся при перезапуске
(
  HANDLE hTimer,                  // хэндл таймера
  const LARGE_INTEGER *pDueTime, // время срабатывания таймера
  LONG lPeriod,                   // период повторения (0 – один раз)
  PTIMERAPCROUTINE pfnCompletionRoutine,  // ук. на асинхр. функцию
  PVOID pvArgToCompletionRoutine, // параметры асинхронной функции
  BOOL bResume // если не 0, выход из спящего состояния
);
```

hTimer определяет нужный таймер. Следующие два параметра (pDueTime и lPeriod) используются совместно. Первый из них задает, когда таймер должен

сработать в первый раз, второй определяет, насколько часто это должно происходить в дальнейшем.

В качестве четвертого и пятого параметров указывается некая асинхронная функция и её параметры, которые нужно реализовать отдельно. Функция должна выглядеть следующим образом:

```
VOID APIENTRY TimerAPCRoutine(PVOID pvArgToCompleUonRoutine, DWORD
dwTimerLowValue, DWORD dwTimerHighValue)
{
  // здесь делаем то, что нужно
}
```

Последний параметр функции lResume полезен на компьютерах с поддержкой режима сна. Обычно в нем передают FALSE, но если передать TRUE, то когда таймер сработает, машина выйдет из режима сна (если она находилась в нем), и пробудятся потоки, ожидавшие этот таймер.

Функция ***CancelWaitableTimer()*** останавливает таймер и отменяет выполнение асинхронной функции, не изменяя состояния таймера.

```
BOOL CancelWaitableTimer(HANDLE hTimer)// остановка таймера
```

6.3. Средства синхронизации потоков в Linux/Unix

В ОС Linux/Unix существуют следующие средства синхронизации:

1) семафоры;
2) мьютексы;
3) условные переменные;
4) блокировки чтения-записи;
5) барьеры;
6) спин-блокировки.

Ниже рассмотрим основные функции для работы с синхронизирующими объектами POSIX. Для получения полного списка функций по каждому объекту рекомендуется воспользоваться дополнительной справочной литературой и специализированными сайтами. В частности, не рассматриваются варианты функций синхронизации с контролем времени, функции для работы с атрибутами объектов и др.

6.3.1 Семафоры

В Unix-системах реализованы три типа **семафоров** – семафоры System V, семафоры POSIX и семафоры в разделяемой памяти. Все объявления функций и типов, относящиеся к POSIX-семафорам, можно найти в файле /usr/include/nptl/semaphore.h.

Семафоры бывают двух типов – именованные и неименованные. Те и другие семафоры хранятся в переменных типа sem_t, но процедура инициализации и уничтожения этих переменных отличается.

Неименованные семафоры инициализируются функцией ***sem_init()***.

```
#include <semaphore.h>
int sem_init(
  sem_t *sem,    // инициализируемый семафор
  int pshared,   // 0, если семафор будет локальным,
                 // ненулевое значение – если семафор
                 // будет разделяемым между процессами
  unsigned int value  // начальное значение флаговой переменной
семафора
);
```

После работы семафор необходимо уничтожить функцией ***sem_destroy()***:

```
#include <semaphore.h>
int sem_destroy(
  sem_t *sem          // уничтожаемый семафор
);
```

Дальнейшая работа с семафором осуществляется с помощью функций ***sem_wait()***, ***sem_trywait()*** и ***sem_post()***:

```
#include <semaphore.h>
int sem_wait(
  sem_t *sem          // указатель на идентификатор семафора
);

int sem_trywait(
  sem_t *sem          // указатель на идентификатор семафора
);

int sem_post(
  sem_t *sem          // указатель на идентификатор семафора
);
```

Эти функции используют указатель на идентификатор семафора, созданного функцией sem_init() и оперируют его значением value. Функция sem_wait() приостанавливает выполнение вызвавшего ее потока до тех пор, пока значение семафора не станет больше нуля, после чего функция уменьшает значение семафора на единицу и возвращает управление. Функция sem_post() увеличивает значение семафора, идентификатор которого был передан ей в качестве параметра, на единицу. Функция sem_trywait() аналогична функции sem_wait(), но если семафор не может быть захвачен, то функция sem_trywait() не устанавливает поток управления в очередь, а возвращает управление с кодом ошибки «семафор не захвачен». В очереди к семафору может находиться одновременно несколько потоков управления. Потоки управления, находящиеся в очереди, упорядочены по приоритетам, а потоки, имеющие равный приоритет, упорядочены по времени установки в очередь.

Получить значение семафора можно функцией ***sem_getvalue()***.

```
#include <semaphore.h>
int sem_getvalue(sem_t *sem, int *sval);
```

Функция получает значение семафора в некоторый неопределенный момент времени. В интервале между исполнением sem_getvalue() и проверкой значения флаговая переменная семафора может измениться, поэтому тот факт, что sem_getvalue() вернул ненулевое значение, не означает, что вызов sem_wait() с этим семафором не будет заблокирован.

Именованные семафоры создаются функцией ***sem_open()***. С её же помощью можно получить доступ к уже существующему именованному семафору. Если процесс попытается несколько раз открыть один и тот же семафор, ему будут возвращать один и тот же указатель.

```
#include <fcntl.h>     /* For O_* constants */
#include <sys/stat.h>  /* For mode constants */
#include <semaphore.h>

sem_t *sem_open(
  const char *name,   // имя семафора
  int oflag,     // флаги (0, O_CREAT и O_CREAT|O_EXC)
  mode_t mode,   // необязательный, используется, только если flags
                 // содержит бит O_CREAT
  unsigned int value  // необязательный, используется только если
                      // flags содержит бит O_CREAT
);
```

Именованные семафоры всегда разделяемые между процессами. При доступе к существующему семафору действует схема проверки прав, аналогичная проверке прав к файлам. Для доступа к семафору процесс должен иметь права чтения и записи.

Для отсоединения от семафора и освобождения памяти используется функция ***sem_close()***:

```
#include <semaphore.h>
int sem_close(
  sem_t *sem          // указатель на идентификатор семафора
);
```

Следует помнить, что закрытие именованного семафора процессом не прекращает существования семафора. Чтобы удалить семафор, необходимо вызвать функцию ***sem_unlink()***. Это лишит новые процессы возможность видеть семафор как существующий и позволит создать новый семафор с тем же именем. Если с семафором на момент вызова функции работали потоки, то семафор останется в рабочем состоянии, пока все процессы не выполнят sem_close().

Во всем остальном именованный семафор не отличается от неименованного. Над ним можно выполнять те же операции, что и над неименованным, при помощи тех же функций.

Рассмотрим пример (см. листинг 6.5). В нем связанный процесс обращается к общему участку памяти, полученному в результате отображения файла на память.

Листинг 6.5. Пример использования семафора для синхронизации процессов:

```
#include <semaphore.h>
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/mman.h>

int main(int argc, char **argv)
{
  int fd, i,count=0,nloop=10,zero=0,*ptr;
  sem_t semaphore;

  //открываем файл и отображаем его в память

  fd = open("log.txt",O_RDWR|O_CREAT,S_IRWXU);
  write(fd,&zero,sizeof(int));
  ptr = mmap(NULL,sizeof(int),PROT_READ
|PROT_WRITE,MAP_SHARED,fd,0);
  close(fd);


  /* создаем семафор */
  if ((semaphore = sem_open("/mysemaphore", O_CREAT, 0644, 1)) ==
SEM_FAILED) {
    perror("semaphore initilization");
    exit(0);
  }

  if (fork() == 0) { /* child process*/
    for (i = 0; i < nloop; i++) {
      sem_wait(&semaphore);
      printf("child: %d\n", (*ptr)++);
      sem_post(&mutex);
    }
    exit(0);
  }
  /* возвращаемся к родительскому процессу */
  for (i = 0; i < nloop; i++) {
    sem_wait(&semaphore);
    printf("parent: %d\n", (*ptr)++);
    sem_post(&semaphore);
```

```
  }
  exit(0);
}
```

Для отображения файла в память используется функция ***mmap()***:

```
#include<sys/mman.h>
void *mmap(
  void *addr,// адрес начала участка отображенной памяти
  size_t len,// количество байт, которое нужно отобразить в память
  int prot, // степень защищённости отображенного участка памяти
  int flag, // атрибуты области
  int filedes, // дескриптор файла, который нужно отобразить
  off_t off // смещение отображенного участка от начала файла
)
```

Возвращает адрес начала участка отображаемой памяти или MAP_FAILED в случае неудачи.

6.3.2 Мьютексы

Все функции и типы данных, имеющие отношение к мьютексам, определены в файле pthread.h. Мьютекс создается вызовом функции ***pthread_mutex_init()***.Перед освобождением памяти из-под мьютекса его необходимо уничтожить функцией ***pthread_mutex_destroy()***. Уничтожение мьютекса без выполнения pthread_mutex_destroy() может приводить к утечке памяти или исчерпанию системных ресурсов. Выполнение операции pthread_mutex_destroy над мьютексом, на котором заблокирован один или более потоков, приводит к неопределенным последствиям.

```
#include <pthread.h>
int pthread_mutex_init(
  pthread_mutex_t *mutex,             // идентификатор мьютекса
  const pthread_mutexattr_t *attr     // атрибуты мьютекса.
);

int pthread_mutex_destroy(
  pthread_mutex_t *mutex   // идентификатор уничтожаемого мьютекса
);
```

Для того чтобы получить исключительный доступ к глобальному ресурсу (захватить мьютекс), поток вызывает функцию ***pthread_mutex_lock()***. Закончив работу с глобальным ресурсом, поток освобождает мьютекс с помощью функции ***pthread_mutex_unlock()***.

```
#include <pthread.h>
int pthread_mutex_lock(
  pthread_mutex_t *mutex   // идентификатор нового мьютекса
```

```
);
int pthread_mutex_trylock(
  pthread_mutex_t *mutex   // идентификатор нового мьютекса
);
int pthread_mutex_unlock(
  pthread_mutex_t *mutex   // идентификатор нового мьютекса
);
```

Если поток вызовет функцию pthread_mutex_lock() для мьютекса, уже захваченного другим потоком, эта функция не вернет управление до тех пор, пока другой поток не освободит мьютекс с помощью вызова pthread_mutex_unlock(). Функция ***pthread_mutex_trylock()*** выполняет те же действия, что и pthread_mutex_lock(), но в случае, если мьютекс занят другим процессом возвращает ошибку.

По умолчанию операции над мьютексами не осуществляют никаких проверок. При повторном захвате мьютекса той же нитью произойдет взаимная блокировка (тупик). Используя pthread_attr_t, при инициализации мьютекса можно задать параметры, которые заставят систему делать проверки при работе с мьютексами и возвращать ошибки при некорректных последовательностях операций, но это повлечет дополнительную нагрузку на операционную систему.

Простейший пример работы с мьютексом приведен ниже (см. листинг 6.6).

Листинг 6.6. Пример использования мьютекса:

```
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <pthread.h>

pthread_mutex_t m;
// глобальная переменная
int count=0;

void* inc_count (void* arg)
{
  pthread_mutex_lock(&m); // сначала захватываем мьютекс
  // выполняем действие над глобальной переменной
  count = count + 1;
  printf("[%d]",count);
  pthread_mutex_unlock(&m); // отпускаем мьютекс
}
int main()
{
  // создаем мьютекс
  pthread_mutex_init(&m,NULL);
  pthread_t t1, t2;

  inc_count(NULL);
```

```
  pthread_create(&t1,NULL,inc_count,NULL); // создаем поток 1

  pthread_create(&t2,NULL,inc_count,NULL); // создаем поток 2

  // ожидаем завершение потоков
  pthread_join(t1,NULL);
  pthread_join(t2,NULL);
  return 0;
}
```

6.3.3 Условные переменные

Процедура создания и уничтожения условных переменных в целом аналогична процедуре создания и уничтожения ранее рассматривавшихся объектов синхронизации. Для создания переменной служит функция ***pthread_cond_init()***, для удаления ***pthread_cond_destroy()***:

```
#include <pthread.h>

int pthread_cond_init(
  pthread_cond_t *cond,     // идентиф. новой условной переменной
  const pthread_condattr_t *attr// атрибуты переменной
);

int pthread_cond_destroy(
  pthread_cond_t *cond      // идентиф. удаляемой усл. переменной
);
```

Как мы помним, над условной переменной определены две основные операции: «ожидание» и «сигнал». Первая функция реализуется с помощью ***pthread_cond_wait()***, вторая – с помощью ***pthread_cond_signal()***:

```
#include <pthread.h>
int pthread_cond_wait(
  pthread_cond_t *cond,     // идентификатор условной переменной
  pthread_mutex_t *mutex    // идентификатор мьютекса
);

int pthread_cond_signal(
  pthread_cond_t *cond      // идентификатор условной переменной
);
```

При вызове pthread_cond_wait() мьютекс, переданный в качестве второго параметра, должен быть захвачен, в противном случае результат не определен. Wait освобождает мьютекс и блокирует поток до момента вызова другим потоком pthread_cond_signal(). После пробуждения wait пытается захватить мьютекс. Если это не получается, он блокируется до того момента, пока мьютекс не освободят. Мьютекс используется для защиты данных, используемых при вычислении условия, с которым связана условная переменная. Условие необходимо про-

верять как перед вызовом pthread_cond_wait(), так и после выхода из этой функции. Проверка условия перед вызовом позволяет защититься от ситуации, когда производитель вызвал signal в то время, когда потребитель еще не был заблокирован в wait. Повторная проверка условия необходима на случай, когда производителей или потребителей несколько и между ними возникает конкуренция.

Листинг 6.7. Пример использования условной переменной:

```
#define _MULTI_THREADED
#include <pthread.h>

#include <stdio.h>

Int conditionmet = 0;
Pthread_cond_t cond  = PTHREAD_COND_INITIALIZER; // усл. Переменная
Pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER; // мьютекс

#define NTHREADS 5

/* Функция проверяет код возврата и завершает программу, если
функция не была выполнена
*/
Static void compresults(char *string, int rc) {
  If (rc) {
    Printf("Ошибка в : %s, rc=%d", string, rc);
    Exit(EXIT_FAILURE);
  }
  Return;
}

Void *threadfunc(void *parm) // функция потока
{
  Int rc;

  Rc = pthread_mutex_lock(&mutex);
  Checkresults("pthread_mutex_lock()\n", rc);

  While (!Conditionmet) {
    Printf("Thread blocked\n");
    Rc = pthread_cond_wait(&cond, &mutex);
    Checkresults("pthread_cond_wait()\n", rc);
  }

  Rc = pthread_mutex_unlock(&mutex);
  Checkresults("pthread_mutex_lock()\n", rc);
  Return NULL;
}

Int main(int argc, char **argv)
{
```

```
  Int rc=0;
  Int i;
  Pthread_t threadid[NTHREADS];

  Printf("Enter Testcase - %s\n", argv[0]);

  Printf("Create %d threads\n", NTHREADS);
  For(i=0; i<NTHREADS; ++i) {
    Rc = pthread_create(&threadid[i], NULL, threadfunc, NULL);
    Checkresults("pthread_create()\n", rc);
  }

  Sleep(5);
  Rc = pthread_mutex_lock(&mutex);
  Checkresults("pthread_mutex_lock()\n", rc);

  /* Условие выполнилось. Устанавливаем флаг и будим все ожидающие
потоки */
  Conditionmet = 1;
  Printf("Wake up all waiting threads...\n");
  Rc = pthread_cond_broadcast(&cond);
  Checkresults("pthread_cond_broadcast()\n", rc);

  Rc = pthread_mutex_unlock(&mutex);
  Checkresults("pthread_mutex_unlock()\n", rc);

  Printf("Wait for threads and cleanup\n");
  For (i=0; i<NTHREADS; ++i) {
    Rc = pthread_join(threadid[i], NULL);
    Checkresults("pthread_join()\n", rc);
  }
  Pthread_cond_destroy(&cond);
  Pthread_mutex_destroy(&mutex);

  Printf("Main completed\n");
  Return 0;
}
```

Обратите внимание, что мы использовали ещё одну функцию ***pthread_cond_broadcast()*** в этой программе. Это широковещательный аналог функции pthread_cond_signal().

6.3.4 Блокировки чтения-записи

API для работы с блокировками чтения-записи в целом похож на API для работы с мьютексами и включает в себя следующие основные функции.

Создание блокировки происходит с помощью функции ***pthread_rwlock_init()***, удаление - ***pthread_rwlock_destroy()***:

```
#include <pthread.h>
```

```
int pthread_rwlock_init(
  pthread_rwlock_t *rwlock,           // идентификатор блокировки
  const pthread_rwlockattr_t *attr    // атрибуты блокировки
);
int pthread_rwlock_destroy(
  pthread_rwlock_t *rwlock       // уничтожаемая блокировка
);
```

Функция pthread_rwlock_init() выделяет ресурсы, необходимые для использования объекта блокировки чтения-записи, адресуемого параметром rwlock, и инициализирует (он переходит в незаблокированное состояние) с использованием объекта атрибутов, адресуемого параметром attr. Если параметр attr содержит значение NULL, для блокировки чтения-записи будут использованы атрибуты, действующие по умолчанию.

Для блокировки чтения используются функции ***pthread_rwlock_rdlock()*** и ***pthread_rwlock_tryrdlock()***.

```
#include <pthread.h>

int pthread_rwlock_rdlock(
  pthread_rwlock_t *rwlock       // идентификатор блокировки
);
int pthread_rwlock_tryrdlock(
  pthread_rwlock_t *rwlock       // идентификатор блокировки
);
```

Для блокировки записи применяются функции ***pthread_rwlock_wrlock()*** и ***pthread_rwlock_trywrlock()***:

```
#include <pthread.h>

int pthread_rwlock_wrlock(
  pthread_rwlock_t *rwlock       // идентификатор блокировки
);
int pthread_rwlock_trywrlock(
  pthread_rwlock_t *rwlock       // идентификатор блокировки
);
```

Варианты ***try*** блокировок выбрасывают исключения, если данная блокировка уже захвачена каким-либо из потоков.

Для снятия блокировки применяется функция ***pthread_rwlock_unlock()***:

```
#include <pthread.h>
int pthread_rwlock_unlock(
  pthread_rwlock_t *rwlock       // идентификатор блокировки
);
```

Если функция вызывается, чтобы освободить блокировку чтения, и существуют другие блокировки чтения, удерживаемые в данный момент по этому объекту блокировки чтения-записи, то он (объект) останется в состоянии блокирования для обеспечения чтения. Если с помощью этой функции освобождается последняя блокировка для чтения по заданному объекту блокировки чтения-записи, то этот объект перейдет в разблокированное состояние и, соответственно, не будет иметь владельцев.

Если эта функция вызывается, чтобы освободить блокировку для обеспечения записи по заданному объекту блокировки чтения-записи, то этот объект перейдет в разблокированное состояние.

Рассмотрим пример использования блокировок. В ней показано как использовать функцию pthread_rwlock_rdlock() для захвата блокировки чтения-записи для чтения.

Листинг 6.8. Пример использования блокировок чтения-записи:

```
#define _MULTI_THREADED
#include pthread.h
#include stdio.h

pthread_rwlock_t rwlock;

static void compResults(char *string, int rc) {
  if (rc) {
    printf("Ошибка в : %s, rc=%d", string, rc);
    exit(EXIT_FAILURE);
  }
  return;
}

void *rdlockThread(void *arg)
{
  int rc;
  printf("Выполняется нить, получение блокировки для чтения\n");
  rc = pthread_rwlock_rdlock(&rwlock);
  compResults("pthread_rwlock_rdlock()\n", rc);
  printf("блокировка rwlock захвачена для чтения\n");
  sleep(5);
  printf("освобождение блокировки для чтения\n");
  rc = pthread_rwlock_unlock(&rwlock);
  compResults("pthread_rwlock_unlock()\n", rc);
  printf("Дополнительная нить разблокировала\n");
  return NULL;
}

void *wrlockThread(void *arg)
{
  int rc;
```

```
  printf("Выполняется нить, получение блокировки для записи\n");
  rc = pthread_rwlock_wrlock(&rwlock);
  compResults("pthread_rwlock_wrlock()\n", rc);

  printf("Блокировка rwlock захвачена для записи, освобождение
блокировки\n");
  rc = pthread_rwlock_unlock(&rwlock);
  compResults("pthread_rwlock_unlock()\n", rc);
  printf("Дополнительная нить разблокирована\n");
  return NULL;
}

int main(int argc, char **argv)
{
  int rc=0;
  pthread_t thread, thread1;

  printf("Запуск теста - %s\n", argv[0]);

  printf("Главная нить, инициализация блокировки чтения-записи\n");
  rc = pthread_rwlock_init(&rwlock, NULL);
  compResults("pthread_rwlock_init()\n", rc);

  printf("Главная нить, захват блокировки для чтения\n");
  rc = pthread_rwlock_rdlock(&rwlock);
  compResults("pthread_rwlock_rdlock()\n",rc);

  printf("Главная нить, повторный захват блокировки для чтения\n");
  rc = pthread_rwlock_rdlock(&rwlock);
  compResults("pthread_rwlock_rdlock() second\n", rc);

  printf("Главная нить, создание для захвата блокировки чтения\n");
  rc = pthread_create(&thread, NULL, rdlockThread, NULL);
  compResults("pthread_create\n", rc);

  printf("Главная нить - освобождение блокировки для чтения\n");
  rc = pthread_rwlock_unlock(&rwlock);
  compResults("pthread_rwlock_unlock()\n", rc);

  printf("Главная нить, создание нити для захвата блокировки для
записи\n");
  rc = pthread_create(&thread1, NULL, wrlockThread, NULL);
  compResults("pthread_create\n", rc);

  sleep(5);
  printf("Главная нить - освобождение второй блокировки для
чтения\n");
  rc = pthread_rwlock_unlock(&rwlock);
  compResults("pthread_rwlock_unlock()\n", rc);
```

```
  printf("Главная нить, ожидание завершения нитей\n");
  rc = pthread_join(thread, NULL);
  compResults("pthread_join\n", rc);

  rc = pthread_join(thread1, NULL);
  compResults("pthread_join\n", rc);

  rc = pthread_rwlock_destroy(&rwlock);
  compResults("pthread_rwlock_destroy()\n", rc);

  printf("Главная нить завершена\n");
  return 0;
}
```

6.3.5 Барьеры

Инициализация и разрушение барьеров происходит с помощью функций ***pthread_barrier_init()***, ***pthread_barrier_destroy()***:

```
#include <pthread.h>
int pthread_barrier_init (
  pthread_barrier_t *restrict barrier,     // идентификатор барьера
  const pthread_barrierattr_t *restrict attr,   // атрибуты барьера
  unsigned count // количество синхронизируемых потоков управления
);

int pthread_barrier_destroy (
  pthread_barrier_t *barrier
);
```

Аргумент count в функции инициализации барьера задает количество синхронизируемых потоков управления. Столько потоков должны вызвать функцию синхронизации на барьере ***pthread_barrier_wait()***, прежде чем каждый из них сможет успешно завершить вызов и продолжить выполнение:

```
#include <pthread.h>
int pthread_barrier_wait (
  pthread_barrier_t *barrier     // идентификатор барьера
);
```

Когда к функции pthread_barrier_wait() обратилось требуемое число потоков управления, одному из них в качестве результата возвращается именованная константа PTHREAD_BARRIER_SERIAL_THREAD, а всем другим выдаются нули. После этого барьер возвращается в начальное (инициализированное) состояние, а выделенный поток может выполнить соответствующие объединительные действия.

Пример программы, использующий барьеры приведен в Листинге 6.9. Барьеры используются для слияния результатов вычислений логарифма по основа-

нию 2, при этом используется два потока. Первый считает ряд для положительных значений, второй – для отрицательных.

Листинг 6.9. Пример программы, использующей барьеры:

```
#define _XOPEN_SOURCE 600

#include <pthread.h>
#include <stdio.h>
#include <errno.h>

static pthread_barrier_t mbrr;

static double sums [2] = {0,0}; // два потока «+» и «-»

static void *start_func (void *ns)
{
    double d = 1;
    double s = 0;
    int i;

    for (i = (int) ns; i <= 100000000; i += 2)
    {
        s += d / i;
    }

    sums [(int) ns - 1] = s;

    /* Синхронизируемся для получения общего итога */
    if (pthread_barrier_wait(&mbrr)==PTHREAD_BARRIER_SERIAL_THREAD)
    {
        sums [0] -= sums [1];
    }
    return (sums); // указатель на итог
}

int main (void)
{
    pthread_t pt1, pt2;
    double *pd;

    if ((errno = pthread_barrier_init(&mbrr, NULL, 2)) != 0)
    {
        perror ("PTHREAD_BARRIER_INIT");
        return (errno);
    }

    pthread_create (&pt1, NULL, start_func, (void *) 1);
    pthread_create (&pt2, NULL, start_func, (void *) 2);
```

```
    pthread_join (pt1,(void **) &pd);
    pthread_join (pt2, (void **) &pd);

    printf ("Совместно вычисленное значение ln(2): %g\n", *pd);

    return (pthread_barrier_destroy(&mbrr));
}
```

6.3.6 Спин-блокировки

Для использования примитивов спин-блокировки необходимо подключить файл <linux/spinlock.h>. Фактическая блокировка имеет тип spinlock_t. Для инициализации спин-блокировки используется функция ***pthread_spin_init()***, для разрушения используется функция ***pthread_spin_destroy()***.

```
#include <pthread.h>

int pthread_spin_init (
  pthread_spinlock_t *lock,// идентификатор блокировки
  int pshared
);

int pthread_spin_destroy (
  pthread_spinlock_t *lock // идентификатор блокировки
);
```

Перед входом в критическую секцию необходимо установить блокировку с помощью функций ***pthread_spin_lock()*** или ***pthread_spin_trylock***, для освобождения полученной блокировки нужно передать её в функцию ***pthread_spin_unlock()***:

```
#include <pthread.h>

int pthread_spin_lock (
  pthread_spinlock_t *lock);

int pthread_spin_trylock (
  pthread_spinlock_t *lock);

int pthread_spin_unlock (
  pthread_spinlock_t *lock);
```

Посмотрим на сколько спин-блокировки работают быстрее, чем мьютексы. В листинге программы 6.10 используются директивы препроцессора #ifdef, #endif, позволяющие указать дополнительную опцию при компиляции программы. Если программа скомпилирована с опцией USE_SPINLOCK используются спин-блокировки, иначе – обычные мьютексы. Программа создает список целых чисел и два потока, которые эти числа из списка удаляют, а средства синхронизации применяются для координации действий потоков. По окончании програм-

мы выводится информация о времени, прошедшем с момента создания потоков до их окончания.

Листинг 6.10. Программа, сравнивающая скорость работы мьютексов и спин-блокировок:

```
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
#include <sys/syscall.h>
#include <errno.h>
#include <sys/time.h>

#include <list>

#define LOOPS 10000000

using namespace std;

list<int> the_list;

#ifdef USE_SPINLOCK // используем директивы препроцессора
pthread_spinlock_t spinlock;
#else
pthread_mutex_t mutex;
#endif

pid_t gettid() { return syscall( __NR_gettid ); }

void *consumer(void *ptr)
{
    int i;
    printf("Consumer TID %lu\n", (unsigned long)gettid());
    while (1)
    {
#ifdef USE_SPINLOCK
        pthread_spin_lock(&spinlock);
#else
        pthread_mutex_lock(&mutex);
#endif

        if (the_list.empty())
        {
#ifdef USE_SPINLOCK
            pthread_spin_unlock(&spinlock);
#else
            pthread_mutex_unlock(&mutex);
#endif
            break;
        }
```

```
        i = the_list.front();
        the_list.pop_front();

#ifdef USE_SPINLOCK
        pthread_spin_unlock(&spinlock);
#else
        pthread_mutex_unlock(&mutex);
#endif
    }

    return NULL;
}

int main()
{
    int i;
    pthread_t thr1, thr2;
    struct timeval tv1, tv2;

#ifdef USE_SPINLOCK
    pthread_spin_init(&spinlock, 0);
#else
    pthread_mutex_init(&mutex, NULL);
#endif

    // Creating the list content...
    for (i = 0; i < LOOPS; i++)
        the_list.push_back(i);

    // Оценка времени между стартом потоков
    gettimeofday(&tv1, NULL);

    pthread_create(&thr1, NULL, consumer, NULL);
    pthread_create(&thr2, NULL, consumer, NULL);

    pthread_join(thr1, NULL);
    pthread_join(thr2, NULL);

    // Оценка времени после выполнения потоков
    gettimeofday(&tv2, NULL);

    if (tv1.tv_usec > tv2.tv_usec)
    {
        tv2.tv_sec--;
        tv2.tv_usec += 1000000;
    }

    printf("Result - %ld.%ld\n", tv2.tv_sec - tv1.tv_sec,
```

```
        tv2.tv_usec - tv1.tv_usec);

#ifdef USE_SPINLOCK
    pthread_spin_destroy(&spinlock);
#else
    pthread_mutex_destroy(&mutex);
#endif

    return 0;
}
```

Скопилируйте программу два раза (с опцией USE_SPINLOCK и без неё) и сравните результаты:

```
[user@host:~]$  g++ -Wall -pthread main.cc
[user@host:~]$ ./a.out
[user@host:~]$  g++ -DUSE_SPINLOCK -Wall -pthread main.cc
[user@host:~]$ ./a.out
```

Задание

1. Изучить краткие теоретические сведения и лекционный материал по теме практического задания.
2. Реализовать все приведенные примеры программ и объяснить результаты их работы.
3. Получить индивидуальное задание у преподавателя и реализовать соответствующие программы с помощью функций WinAPI и POSIX API.
4. Сравнить возможности обоих подходов, сделать выводы.

Контрольные вопросы

1. Что такое взаимоблокировка? Какие стратегии борьбы с взаимоблокировками Вам известны?
2. Что собой представляет бесконечная отсрочка? Какие стратегии борьбы с бесконечной отсрочкой Вам известны?
3. Что такое гонка данных? Какие стратегии борьбы с гонкой данных Вам известны?
4. Что такое семафор? Каких проблем синхронизации позволяет избежать семафор?
5. Какие типы семафоров есть в ОС Linux/Unix? Чем отличаются именованные семафоры от неименованных?
6. Что такое мьютекс? Каких проблем синхронизации позволяет избежать мьютекс?
7. Что такое условная переменная? Каких проблем синхронизации позволяет избежать условная переменная?
8. Что такое критическая секция? Каких проблем синхронизации позволяет избежать условная переменная?

9. Что такое блокировка чтения-записи? Каких проблем синхронизации позволяет избежать блокировка чтения-записи?

10. Что такое барьер? Каких проблем синхронизации позволяет избежать барьер?

11. Что такое спин-блокировка? Каких проблем синхронизации позволяет избежать спин-блокировка?

12. Что такое ожидающий таймер и для чего он используется?

13. Какие средства синхронизации существуют в ОС Windows?

14. Какие средства синхронизации существуют в ОС Linux/Unix?

15. Зачем нужны директивы препроцессора? Какие директивы препроцессора Вы знаете?

16. Что такое отображения файлов? Какие функции существуют для отображения файла на память?

Варианты заданий

См. задание по Теме №5. Предусмотреть в варианте задания использование разделяемого ресурса. Описать какие из средств синхронизации и как могут быть применены для решения новой задачи. Реализовать один (или несколько) из предложенных Вами вариантов для ОС Windows и Linux/Unix.

Тема № 7

Программирование сокетов

Цель работы

Познакомиться с основными аспектами работы с сокетами. Познакомиться с соответствующими функциями WinAPI и POSIX API.

Краткие теоретические сведения

7.1. Общие сведения о сокетах

Сокеты – одно из средств межпроцессного взаимодействия (IPC). Сокет представляет собой один конец двусторонней связи, между двумя программами. Соединяя два сокета, можно передавать данные между разными процессами как в рамках одной системы, так и между процессами, запущенными на разных машинах в сети. Кроме того, с их помощью можно организовать взаимодействие с программами, работающими под управлением других операционных систем.

Реализация сокетов осуществляет инкапсуляцию протоколов сетевого и транспортного уровней.

Сокет состоит из IP адреса машины и номера порта, используемого приложением. Уникальность IP адресов в сети Интернет и порта в рамках одной машины делает сам сокет уникальным в сети Интернет.

При взаимодействии двух процессов, использующих сокеты, можно выделить серверную часть и клиентскую: серверный процесс инициализирует сокет и ждёт входящих соединений от других процессов, а клиентский процесс инициализирует соединение с сервером.

Существует два типа сокетов **потоковые** и **датаграммные**.

Потоковый сокет – это сокет с установлением соединения, состоящий из потока байтов, который может быть двунаправленным, то есть через конечную точку может передавать и получать данные. Потоковый сокет осуществляет надежную передачу, подходит для передачи больших объемов данных. Для передачи используется протокол TCP.

Датаграммные сокеты – сокеты без установления соединения. Используется протокол UDP. По сравнению с потоковым сокетом обмен данными происходит быстрее, но является ненадёжным: сообщения могут теряться в пути, дублироваться и переупорядочиваться.

В случае **локального** варианта взаимодействия через сокеты обмен данными происходит через специальные файлы, расположенные в файловой системе и, фактически, является расширением идеи именованных каналов, но с интерфейсом сокетов.

Рассмотрим, какие средства для работы с сокетами существуют в операционных системах Unix/Linux

7.2. Потоковые сокеты

Алгоритм работы клиентского и серверного приложений, использующих потоковые сокеты, и соответствующие API функции представлены на рис. 7.1. Рассмотрим этапы и функции подробнее.

Все объявления функций и типов, относящиеся к сокетам, можно найти в файле socket.h.

Для создания сокета используется функция ***socket()***:

```
#include <sys/types.h>
#include <sys/socket.h>
int socket (     // создание сокета
  int domain,    // домен и семейство адресов: AF_UNIX и AF_INET
  int type,      // тип сокета: SOCK_STREAM, SOCK_DGRAM, SOCK_RAW
  int protocol   // протокол
);
```

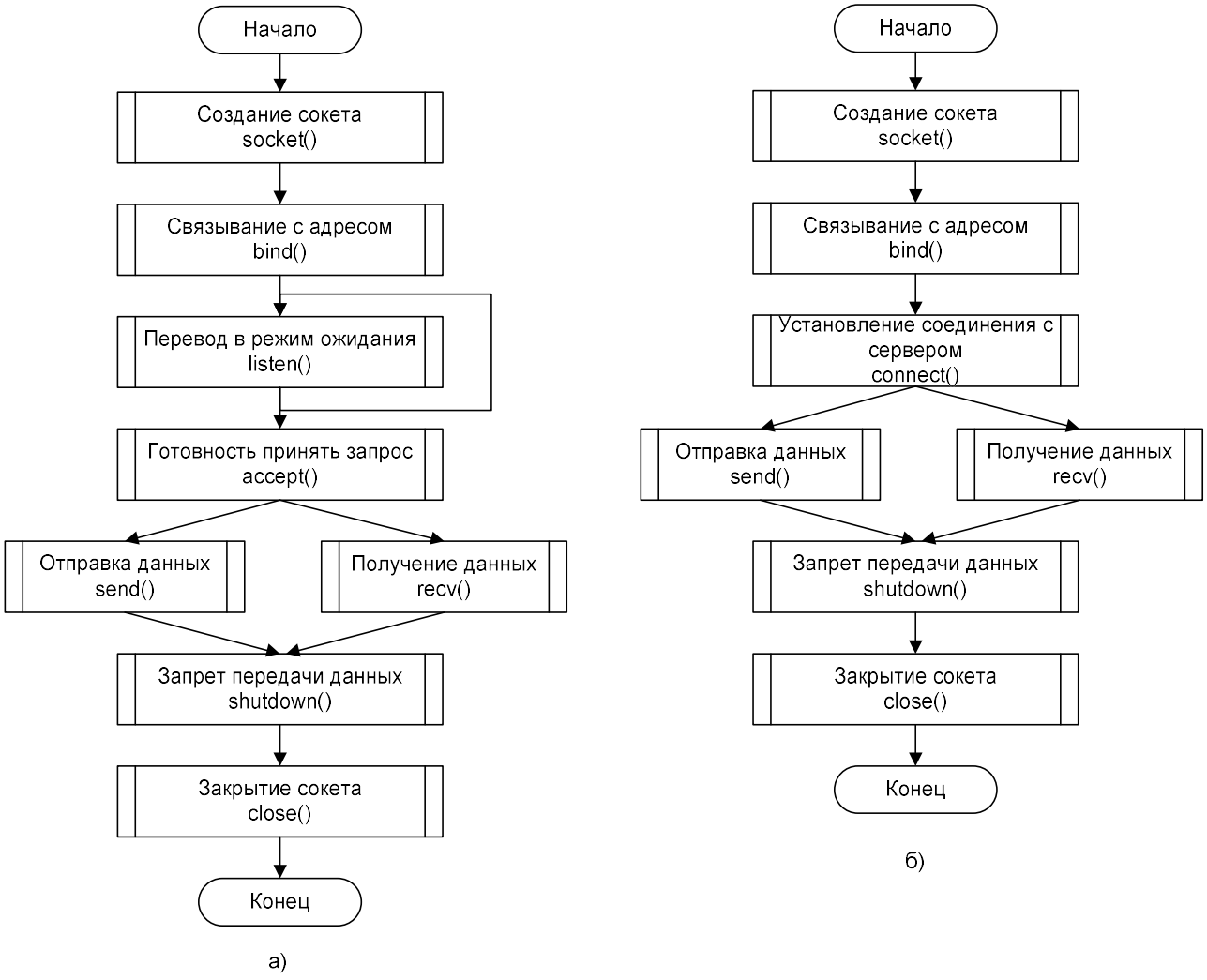

Рис. 7.1. Алгоритм установления связи и обмена данными между клиентом (б) и сервером (а) на основе потокового сокета

Сокет имеет три атрибута:

1) домен - чаще других используются домены Unix и Internet, которые задаются константами AF_UNIX, AF_INET (для протокола IPv4) и AF_INET6 (для протокола IPv6) соответственно.

2) тип – определяет способ передачи данных по сети. SOCK_STREAM – потоковый сокет, SOCK_DGRAM – датаграммный, SOCK_RAW используется для низкоуровневой работы с протоколами IP, ICMP.

3) протокол – как правило, определяется автоматически в зависимости от типа сокета и домена - в этом случае передается 0.

Для связи сокета с адресом используется функция ***bind()***:

```
#include <sys/types.h>
#include <sys/socket.h>
int bind ( // связывание сокета сервера с адресом
  int sockfd, // дескриптор сокета
  struct sockaddr *addr, // указатель на структуру с адресом
  int addrlen // длина структуры адреса
);
```

Первый параметр функции - это дескриптор сокета, который мы хотим привязать к заданному адресу. Второй параметр содержит указатель на структуру с адресом, а третий - длину этой структуры.

Процедуры создания сокета и связывания его с адресом являются общими для клиента и для сервера. Перейдем к рассмотрению специфики.

Серверный сокет нужно перевести в режим ожидания соединений от клиентов. Делается это с помощью функции ***listen()***, принимающей на входе дескриптор очереди и размер очереди запросов:

```
int listen ( // перевод сокета сервера в режим ожидания
              //запросов со стороны клиентов
  int sockfd, // дескриптор сокета
  int backlog // размер очереди запросов
);
```

В случае если клиентский сокет пытается соединиться с серверным, а у последнего вся очередь заполнена, запрос игнорируется.

Когда сервер готов принять запрос, он вызывает функцию ***accept()***:

```
int accept ( // сервер создаёт для общения с клиентом
             // новый сокет, когда сервер готов принять запрос
  int sockfd, // дескриптор слушающего сокета
  void *addr, // адрес сокета клиента
  int *addrlen // длина
);
```

Функция создает новый сокет и возвращает дескриптор этого сокета. При этом адрес нового сокета такой же, как у слушающего сокета.

Установка соединения со стороны клиента инициализируется с помощью функции ***connect()***:

```
int connect ( //  установления соединения клиентов с сервером
  int sockfd, // сокет для обмена данными с сервером
  struct sockaddr *serv_addr, // ук. на структуру с адресом сервера
  int addrlen // длина этой структуры
);
```

Первый параметр задает сокет, который будет использоваться для обмена данных с сервером, второй – указатель на структуру данных, содержащую адрес сервера, третий – длина этой структуры. Клиентскому сокету можно предварительно присвоить порт, вызвав функцию bind(), в противном случае порт будет выбран автоматически.

После того как инициализированы клиентский и серверный сокеты, можно начинать обмен данными. Для этого предназначены функции ***send()*** (отправка данных) и ***recive()*** (получение данных):

```
int send (       // отправка данных
  int sockfd,    // дескриптор сокета для отправки данных
  const void *msg, // указатель на буфер с данными
  int len,       // длина буфера данных
  int flags      // набор битовых флагов
);

int recv(   // чтение данных из сокета
int sockfd, // дескриптор сокета, из которого считываются данные
void *buf,  // указатель на буфер
int len,    // длина буфера данных
int flags   // набор битовых флагов
);
```

Обе функции получают на входе дескриптор сокета, указатель на буфер и набор флагов. Функция send() возвращает количество отправленных данных (в байтах) или -1 в случае ошибки, recv() возвращает количество полученных данных (в байтах), 0 в случае разрыва соединения и -1 в случае ошибки.

Запретить передачу данных в одном направлении можно с помощью функции ***shutdown()***, конкретизировав что именно вы хотите запретить (чтение, запись, чтение и запись) с помощью параметра how:

```
int shutdown (// запрет передачи данных в одном направлении
  int sockfd, // дескриптор сокета
  int how     // 0 – запрет чтения, 1 – записи, 2 – чтения и записи.
);
```

Закрытие сокета происходит с помощью ***close()***:

```
int close (// закрытие сокета
```

```
  int fd // дескриптор сокета
);
```

7.3. Датаграммные сокеты

Процедуры создания сокета и связывания для датаграммного и потокового сокета идентичны. Создав датаграммный сокет, его можно сразу использовать для отправки и получения данных. Соединение устанавливать не требуется. Закрытие датаграммного сокета осуществляется функцией ***close()*** (см. рисунок 7.2).

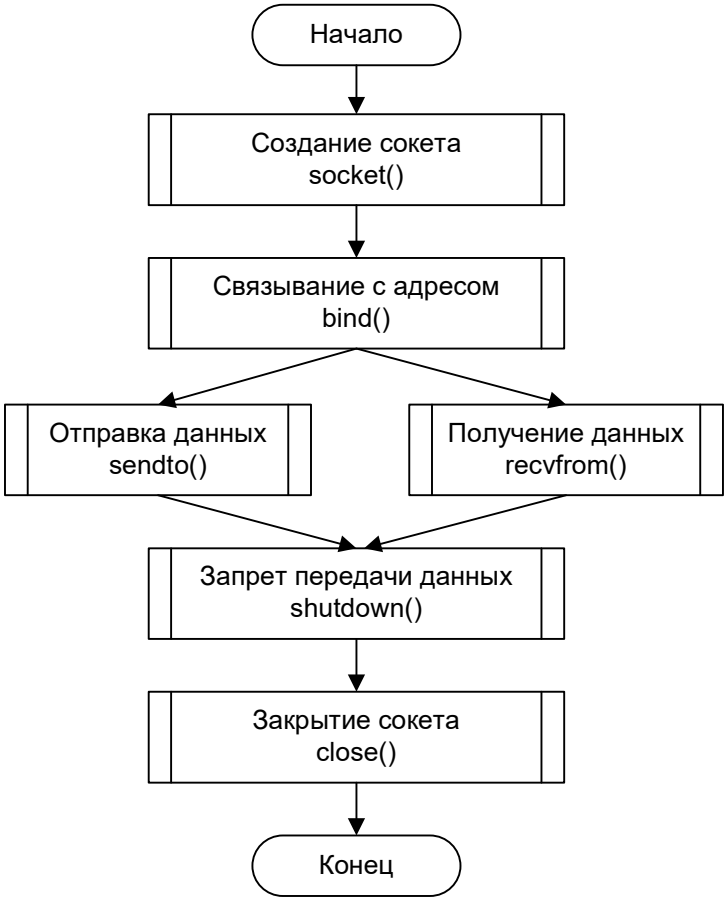

Рис. 7.2. Алгоритм работы датаграммного сокета

Для отправки данных используется функция ***sendto()***, для получения данных используется функция ***recvfrom()***. Параметры функций:

```
int sendto (          // отправка данных через датаграммный сокет
  int sockfd,         // дескриптор сокета
  const void *msg,    // указатель на буфер с данными
  int len,            // длина буфера данных
  unsigned int flags, // набор битовых флагов
  const struct sockaddr *to,    // указатель на структуру с адресом
  int tolen                // длина структуры адреса
);

int recvfrom (        // получение данных из датаграммного сокета
  int sockfd,         // дескриптор сокета
```

```
  void *buf,          // указатель на буфер с данными
  int len,            // длина буфера данных
  unsigned int flags, // набор битовых флагов
  struct sockaddr *from,// указатель на структуру с адресом
  int *fromlen        // длина структуры адреса
);
```

7.4. Примеры использования сокетов

Пример простого клиент-серверного приложения для интернет домена на основе потоковых сокетов приведен в листинге 7.1 (сервер) и листинге 7.2 (клиент).

Листинг 7.1. Пример простого TCP сервера в интернет домене:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

void error(const char *msg)
{
    perror(msg);
    exit(1);
}

int main(int argc, char *argv[])
{
     int sockfd, newsockfd, portno;
     socklen_t clilen;
     char buffer[256];
     struct sockaddr_in serv_addr, cli_addr;
     int n;
     if (argc < 2) {
         fprintf(stderr,"ERROR, no port provided\n");
         exit(1);
     }
     sockfd = socket(AF_INET, SOCK_STREAM, 0);
     if (sockfd < 0)
        error("ERROR opening socket");
     bzero((char *) &serv_addr, sizeof(serv_addr));
     portno = atoi(argv[1]);
     serv_addr.sin_family = AF_INET;
     serv_addr.sin_addr.s_addr = INADDR_ANY;
     serv_addr.sin_port = htons(portno);
     if (bind(sockfd, (struct sockaddr *) &serv_addr,
              sizeof(serv_addr)) < 0)
              error("ERROR on binding");
```

```
     listen(sockfd,5);
     clilen = sizeof(cli_addr);
     newsockfd = accept(sockfd, (struct sockaddr *) &cli_addr,
                        &clilen);
     if (newsockfd < 0)
          error("ERROR on accept");
     bzero(buffer,256);
     n=recv(newsockfd,buffer,255, 0);

     if (n < 0) error("ERROR reading from socket");
     printf("Here is the message: %s\n",buffer);
     n=send(newsockfd, "I got your message", 18, 0);
     if (n < 0) error("ERROR writing to socket");
     close(newsockfd);
     close(sockfd);
     return 0;
}
```

Листинг 7.2. Пример простого TCP клиента в интернет домене:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

void error(const char *msg)
{
    perror(msg);
    exit(0);
}

int main(int argc, char *argv[])
{
    int sockfd, portno, n;
    struct sockaddr_in serv_addr;
    struct hostent *server;

    char buffer[256];
    if (argc < 3) {
       fprintf(stderr,"usage %s hostname port\n", argv[0]);
       exit(0);
    }
    portno = atoi(argv[2]);
    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd < 0)
        error("ERROR opening socket");
```

```
    server = gethostbyname(argv[1]);
    if (server == NULL) {
        fprintf(stderr,"ERROR, no such host\n");
        exit(0);
    }
    bzero((char *) &serv_addr, sizeof(serv_addr));
    serv_addr.sin_family = AF_INET;
    bcopy((char *)server->h_addr,
         (char *)&serv_addr.sin_addr.s_addr,
         server->h_length);
    serv_addr.sin_port = htons(portno);
    if (connect(sockfd,(struct sockaddr *)
&serv_addr,sizeof(serv_addr)) < 0)
        error("ERROR connecting");
    printf("Please enter the message: ");
    bzero(buffer,256);
    fgets(buffer,255,stdin);
    n=send(sockfd, "I got your message", 18, 0);
    if (n < 0)
         error("ERROR writing to socket");
    bzero(buffer,256);
    n=recv(sockfd,buffer,255, 0);
    if (n < 0)
         error("ERROR reading from socket");
    printf("%s\n",buffer);
    close(sockfd);
    return 0;
}
```

Программы нужно откомпилировать каждую отдельно и запустить. В качестве параметра серверу передать номер порта от 2000 до 65535:

```
$ ./server 5555
```

Для запуска клиента потребуется указать имя компьютера в сети и порт. Например:

```
$ ./client computer 5555
```

После ввода сообщения на клиентской машине, оно должно отобразиться на серверной.

В случае если программы работают на одной машине, необходимо использовать UNIX домен (AF_UNIX) и соответствующую структуру для адреса:

```
Struct sockaddr_un
{
 short sun_family;    // AF_UNIX
 char sun_path[108];  // стандартная форма пути для файловой системы
};
```

В этом случае сокет идентичен именованному каналу (FIFO pipe) и выглядит в системе как файл нулевой длины.

Полный пример сервера и клиента представлен в листингах 3 и 4 соответственно.

Листинг 7.3. Пример простого TCP сервера в UNIX домене:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/un.h>
#include <stdio.h>
void error(const char *);
int main(int argc, char *argv[])
{
   int sockfd, newsockfd, servlen, n;
   socklen_t clilen;
   struct sockaddr_un  cli_addr, serv_addr;
   char buf[80];

   if ((sockfd = socket(AF_UNIX,SOCK_STREAM,0)) < 0)
       error("creating socket");
   bzero((char *) &serv_addr, sizeof(serv_addr));
   serv_addr.sun_family = AF_UNIX;
   strcpy(serv_addr.sun_path, argv[1]);
   servlen=strlen(serv_addr.sun_path) +
                    sizeof(serv_addr.sun_family);
   if(bind(sockfd,(struct sockaddr *)&serv_addr,servlen)<0)
       error("binding socket");

   listen(sockfd,5);
   clilen = sizeof(cli_addr);
   newsockfd = accept(
        sockfd,(struct sockaddr *)&cli_addr,&clilen);
   if (newsockfd < 0)
        error("accepting");
   n=read(newsockfd,buf,80);
   printf("A connection has been established\n");
   write(1,buf,n);
   write(newsockfd,"I got your message\n",19);
   close(newsockfd);
   close(sockfd);
   return 0;
}

void error(const char *msg)
{
    perror(msg);
```

```
    exit(0);
}
```

Листинг 7.4. Пример простого TCP клиента в Unix домене:

```
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <stdio.h>
void error(const char *);

int main(int argc, char *argv[])
{
   int sockfd, servlen,n;
   struct sockaddr_un  serv_addr;
   char buffer[82];

   bzero((char *)&serv_addr,sizeof(serv_addr));
   serv_addr.sun_family = AF_UNIX;
   strcpy(serv_addr.sun_path, argv[1]);
   servlen = strlen(serv_addr.sun_path) +
                sizeof(serv_addr.sun_family);
   if ((sockfd = socket(AF_UNIX, SOCK_STREAM,0)) < 0)
       error("Creating socket");
   if (connect(sockfd, (struct sockaddr *)
                        &serv_addr, servlen) < 0)
       error("Connecting");
   printf("Please enter your message: ");
   bzero(buffer,82);
   fgets(buffer,80,stdin);
   write(sockfd,buffer,strlen(buffer));
   n=read(sockfd,buffer,80);
   printf("The return message was\n");
   write(1,buffer,n);
   close(sockfd);
   return 0;
}

void error(const char *msg)
{
    perror(msg);
    exit(0);
}
```

Обратите внимание, что в двух последних примерах используются системные вызовы ***read()*** и ***write()*** для чтения и записи в сокет. Этот способ также допустим, но имеет ряд ограничений, поэтому злоупотреблять им не стоит.

Пример датаграммного сервера и простого UDP клиента для интернет домена представлен в листингах 7.5 и 7.6.

Листинг 7.5. Пример датаграммного сервера:

```
#include <sys/types.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <string.h>
#include <netdb.h>
#include <stdio.h>

void error(const char *msg)
{
    perror(msg);
    exit(0);
}

int main(int argc, char *argv[])
{
   int sock, length, n;
   socklen_t fromlen;
   struct sockaddr_in server;
   struct sockaddr_in from;
   char buf[1024];

   if (argc < 2) {
      fprintf(stderr, "ERROR, no port provided\n");
      exit(0);
   }

   sock=socket(AF_INET, SOCK_DGRAM, 0);
   if (sock < 0) error("Opening socket");
   length = sizeof(server);
   bzero(&server,length);
   server.sin_family=AF_INET;
   server.sin_addr.s_addr=INADDR_ANY;
   server.sin_port=htons(atoi(argv[1]));
   if (bind(sock,(struct sockaddr *)&server,length)<0)
       error("binding");
   fromlen = sizeof(struct sockaddr_in);
   while (1) {
       n = recvfrom(sock,buf,1024,0,(struct sockaddr
*)&from,&fromlen);
       if (n < 0) error("recvfrom");
       write(1,"Received a datagram: ",21);
       write(1,buf,n);
```

```
        n = sendto(sock,"Got your message\n",17,
                   0,(struct sockaddr *)&from,fromlen);
        if (n  < 0) error("sendto");
    }
    return 0;
 }
```

Листинг 7.6. Пример простого UDP-клиента:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

void error(const char *);
int main(int argc, char *argv[])
{
    int sock, n;
    unsigned int length;
    struct sockaddr_in server, from;
    struct hostent *hp;
    char buffer[256];

    if (argc != 3) { printf("Usage: server port\n");
                     exit(1);
    }
    sock= socket(AF_INET, SOCK_DGRAM, 0);
    if (sock < 0) error("socket");

    server.sin_family = AF_INET;
    hp = gethostbyname(argv[1]);
    if (hp==0) error("Unknown host");

    bcopy((char *)hp->h_addr,
          (char *)&server.sin_addr,
           hp->h_length);
    server.sin_port = htons(atoi(argv[2]));
    length=sizeof(struct sockaddr_in);
    printf("Please enter the message: ");
    bzero(buffer,256);
    fgets(buffer,255,stdin);
    n=sendto(sock,buffer,
             strlen(buffer),0,(const struct sockaddr
*)&server,length);
    if (n < 0) error("Sendto");
```

```
   n = recvfrom(sock,buffer,256,0,(struct sockaddr *)&from,
&length);
   if (n < 0) error("recvfrom");
   write(1,"Got an ack: ",12);
   write(1,buffer,n);
   close(sock);
   return 0;
}

void error(const char *msg)
{
    perror(msg);
    exit(0);
}
```

Параметры запуска примеров программ 7.5 и 7.6 аналогичны тем, что использовались при запуске 7.1 и 7.2 программ.

Для того, чтобы посмотреть состояние всех сокетов в системе, можно воспользоваться системной командой ***netstat***:

```
$ netstat –an
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 0.0.0.0:10050           0.0.0.0:*               LISTEN
tcp        0      0 127.0.0.1:199           0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:27017           0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:3306            0.0.0.0:*               LISTEN
tcp        0      0 192.168.147.120:59899   192.168.147.3:1031      ESTABLISHED
tcp        0      0 192.168.147.120:10050   192.168.147.149:43103   TIME_WAIT
tcp        0      0 192.168.147.120:10050   192.168.147.149:43091   TIME_WAIT
tcp        0      0 192.168.147.120:10050   192.168.147.149:42323   TIME_WAIT
Active UNIX domain sockets (servers and established)
Proto RefCnt Flags       Type       State         I-Node Path
unix  2      [ ACC ]     STREAM     LISTENING     8547   /tmp/.winbindd/pipe
unix  2      [ ACC ]     STREAM     LISTENING     8549
unix  22     [ ]         DGRAM                    7479   /dev/log
unix  2      [ ACC ]     STREAM     LISTENING     7554   /var/run/acpid.socket
unix  2      [ ]         DGRAM                    1493   @/org/kernel/udev/udevd
unix  3      [ ]         STREAM     CONNECTED     11872246
unix  2      [ ]         DGRAM                    11871524
unix  3      [ ]         STREAM     CONNECTED     11871396
```

Задание

1. Изучить краткие теоретические сведения и лекционный материал по теме практического задания.
2. Реализовать приведенные примеры программ.
3. Самостоятельно изучить средства программирования сокетов в ОС Windows и отразить в отчете основные API функции.
4. Реализовать примеры клиентских программ под ОС Windows для обмена сообщениями с серверами TCP и UDP для Unix/Linux.
5. Написать отчет и защитить у преподавателя.

Варианты заданий

Отсутствуют.

Контрольные вопросы

1. Что такое сокет?
2. Какие бывают сокеты, в чем их особенности?
3. Какие атрибуты есть у сокета?
4. Особенность приложения клиент – сервер, основанного на потоковом сокете?
5. Алгоритм установления связи между клиентом и сервером для взаимодействия на основе потокового сокета.
6. Алгоритм работы датаграммного сокета.
7. Как завершить соединение между клиентом и сервером?
8. Какие сокеты участвуют при взаимодействии приложений?
9. Как осуществляется прием и отправка данных между клиентом и сервером?

Вопросы к контрольным работам, зачету и экзамену

1. Программа. Программное обеспечение. Отличие программ от ПО. Необходимые свойства ПО.

2. Объекты ядра операционных систем. Таблица описателей объектов ядра. Учет поль-зователей объекта ядра. Дескриптор защиты.

3. Сокеты как средство межпроцессного взаимодействия. Атрибуты сокета. Виды соке-тов.

4. Системные, прикладные, промежуточные программы. Системные управляющие и системные обслуживающие программы.

5. Совместное использование объектов ядра. Наследование описателя объекта. Имено-ванные объекты. Дублирование описателей объектов.

6. Алгоритмы работы потоковых и датаграммных сокетов.

7. Операционная система и операционная среда, система программирования. Современ-ные тенденции развития ПО.

8. Многозадачность. Пакетная обработка. Системы разделения времени. Системы реаль-ного времени. Мультипроцессорная обработка.

9. Способы организации циклов в Ассемблере. Организация подпрограмм в Ассемблере.

10. Этапы разработки программного обеспечения и требования к ПО на этих этапах.

11. Задания, процессы, потоки и волокна.

12. ОС Unix и Linux. История создания. Основные дистрибутивы. Назначение ОС Unix и Linux.

13. Исходный, объектный, загрузочный модули.

14. Процессы. Адресное пространство процессов. Образ процесса. Создание и завершение процессов.

15. Монолитное ядро, микроядро, экзоядро операционной системы.

16. Трансляция и трансляторы. Этапы трансляции. Компиляция и интерпретация.

17. Модели процессов. Состояния процесса. Особенности процессов в Windows и UNIX.

18. Структура каталогов ОС Linux.

19. Загрузчик. Функции загрузчика. Абсолютный загрузчик и абсолютные программы. Связывающий загрузчик.

20. Многопоточность. Отличия от многозадачности. Преимущества и недостатки исполь-зования многопоточности.

21. Пользователи и группы ОС Linux.

22. Кросс-системы.

23. Поток. Модели потоков. Реализация потоков в пространстве пользователя и в ядре, преимущества и недостатки.

24. Объекты файловой системы ОС Linux и права.

25. Принципы Фон Неймана. Архитектура с общей шиной, достоинства и недостатки.

26. Модели построения многопоточных приложений.

27. Программы в ОС Linux. Установка, запуск программ.

28. Регистры процессора. Регистры общего назначения.

29. Состояния потоков. Особенности работы с потоками в Windows и UNIX.

30. Демоны в ОС Linux.

31. Стек. Регистр стека. Индексные регистры.

32. Планирование потоков и процессов. Алгоритмы планирования потоков: статические и динамические; вытесняющие, невытесняющие и смешанные; краткосрочные, среднесрочные, долгосрочные, ввода-вывода.

33. Механизмы безопасности в ОС Linux.

34. Регистр командного указателя. Сегментные регистры.

35. Основные проблемы синхронизации параллельно выполняющихся процессов и пото-ков.

36. Механизмы безопасности в ОС Windows.

37. Флаговый регистр. Системные, управляющие флаги и флаги состояния.

38. Взаимоблокировки. Условия возникновения, стратегии борьбы с взаимоблокировка-ми. Средства синхронизации для решения проблемы взаимоблокировок.

39. Качество ПО. Модель качества ПО. Характеристики качества ПО.

40. Директивы сегментации в ассемблере, упрощенные директивы сегментации.

41. Бесконечная отсрочка. Условия возникновения и стратегии борьбы с бесконечной от-срочкой. Средства синхронизации для решения проблемы бесконечной отсрочки.

42. Тестирование ПО. Уровни тестирования ПО.

43. Методы адресации. Прямая, непосредственная, косвенная, автоинкрементная, регист-ровая, относительная адресация.

44. Гонка данных. Условия возникновения и стратегии борьбы с гонкой данных. Средства синхронизации для решения проблемы гонки данных.

45. Классификация средств защиты ПО.

46. Прерывания. Внешние, внутренние и программные прерывания. Маскируемые и не-маскируемые прерывания.

47. Мьютексы, фьютексы.

48. Методы защиты ПО.

49. Обработка прерывания. Обработчик прерывания. Точные и неточные прерывания. Приоритезация.

50. Критические секции, ожидающие таймеры.

51. Критерии защиты средств ПО.

52. Вектор прерывания. Таблица векторов прерываний. Дескрипторная таблица прерыва-ний.

53. Блокировки чтения-записи, спин-блокировки.

54. Электронные ключи и программные замки как средство защиты ПО от несанкциони-рованного доступа.

55. Нарушения, ловушки, аварии. Обработка в защищенном режиме.

56. Способы межпроцессорного взаимодействия.

57. Средства защиты ПО от несанкционированного копирования.

58. Перехват прерываний. Реентерабельность.

59. Потоки ввода, вывода и ошибок.

60. Парольная защита как средство защиты ПО от несанкционированного доступа.

61. Основные группы команд языка Ассемблер.

62. Каналы. Неименованные и именованные каналы.

63. Условные переменные.

64. Способы передачи параметров в процедуры при связывании разноязыковых модулей программы.

65. Сигналы как средство межпроцессорного взаимодействия.

66. Показатели применимости средств защиты ПО: технические, экономические, органи-зационные.

67. Системные вызовы. Требования к реализации системных вызовов.

68. Переменные окружения процесса и системы.

69. Организация массивов, структур, записей в Ассемблере.

70. Обработка системных вызовов. Централизованная и децентрализованная схема обра-ботки системных вызовов. Диспетчер системных вызовов.

71. Разделяемая память как средство межпроцессного взаимодействия.

72. Барьеры.

73. API функции. Классификация API функций. Место API функций в программировав-нии. WinAPI, POSIX API.

74. Семафоры.

75. Отображение файла/устройства на память.

Темы индивидуальных заданий для самостоятельной работы

Задание выполняется студентами по конкретной теме, которую следует изучить самостоятельно на основе литературных данных и материалов сети Интернет.

Работа над индивидуальным заданием позволяет приобрести определенные навыки в программировании, обобщении и изложении материала по интересующим студента вопросам, а также навыки оформления материала.

Задание выполняется на листах формата А4 и должно включать титульный лист, оглавление, введение, основную часть, заключение, список использованных источников и приложения. Задание должно быть выполнено в соответствии с общими требованиями и правилами оформления, принятыми в университете.

При оценке учитывается полнота раскрытия темы, актуальность представленного материала, соответствие общим требованиям и правилам оформления.

Темы индивидуальных заданий:

1. Программирование для ОС Android.
2. Программирование для ОС Windows Phone.
3. Программирование для ОС Apple iOS.
4. Программирование для ОС Blackberry OS.
5. Программирование для ОС Firefox OS.

Литература

1. Молчанов А.Ю. Системное программное обеспечение. - Учебник для вузов. - 3-е изд. - СПб.: Питер, 2010. - 400 с.: ил.

2. Юров В.И. Assembler. Учебник для вузов. 2-е изд. - СПб.: Питер, 2003. - 637 с.

3. Таненбаум Э. Современные операционные системы (3-е издание). - СПб.: Питер, 2010. - 1120 с.

4. Лав Роберт Linux. Системное программирование (2-е изд.). - Питер, 2014. - 448 с.

5. Джонсон М. Харт. Системное программирование в среде Windows (3-е издание). - Пер. с англ. - М.: Издательский дом "Вильямс", 2005. – 592 с.

6. Побегайло А.П. Системное программирование в Windows : Наиболее полное руководство / А. П. Побегайло. - СПб. : БХВ-Петербург, 2006. – 1055 с. : портр., табл., ил. эл. опт. диск (CD-ROM).

7. Дьяконов В.Ю. Системное программирование : Учебное пособие для втузов / Владимир Юрьевич Дьяконов, Владимир Анатольевич Китов, Игорь Алексеевич Калинчев; Ред. А. Л. Горелик. - М. : Высшая школа, 1990. - 220 с. : ил, табл.

8. Одиноков В.В. Операционные системы и сети : учебное пособие / В. В. Одиноков, В. П. Коцубинский ; Федеральное агентство по образованию, Томский государственный университет систем управления и радиоэлектроники. - 2-е изд., доп. - Томск : ТУСУР, 2008. - 389 с. : ил.

9. Раводин О.М. Операционные системы : Учебное пособие / О. М. Раводин, В. О. Раводин ; Министерство образования Российской Федерации, Томский государственный университет систем управления и радиоэлектроники, Кафедра комплексной информационной безопасности электронно-вычислительных систем. - 2-е изд., перераб. и доп. - Томск : В-Спектр, 2007. - 165 с. : ил.

10. Боровский А. Программирование для Linux [Электронный ресурс]. - Режим доступа: http://citforum.ru/programming/unix/borovsky/, свободный.

11. Стивенс У. UNIX: взаимодействие процессов. - СПб.: Питер, 2002. - 624 с.

12. Гунько А.В. Системное программное обеспечение [Электронный ресурс]. - 2008. - Режим доступа: gun.cs.nstu.ru/ssw/labs.doc, свободный.

13. Sockets Tutorial [Электронный ресурс]. - Режим доступа: http://www.linuxhowtos.org/C_C++/socket.htm, свободный.

14. Деревянко А.С. Конспект лекций по курсу "Системное программирование" [Электронный ресурс]. - Режим доступа: http://khpi-iip.mipk.kharkiv.edu/library/sp/sp2/index.html, свободный.

15. Иванова Г.С., Ничушкина Т.Н. Программирование на ассемблере MASM32 в среде RADAsm с использованием 32-разрядного отладчика OlleDBG

[Электронный ресурс]. - 2010. - Режим доступа: http://e-learning.bmstu.ru/moodle/file.php/1/common_files/library/SPO/Lab1_2/bmstu_iu6_Sysprogr_lab_1_2.pdf, свободный.

16. Гриценко, Ю.Б. Операционные системы. Ч.1. [Электронный ресурс] : учеб. пособие — Электрон. дан. — М. : ТУСУР, 2009. — 187 с. — Режим доступа: http://e.lanbook.com/book/4972 — Загл. с экрана.

17. Гриценко, Ю.Б. Операционные системы. Ч.2. [Электронный ресурс] : учеб. пособие — Электрон. дан. — М. : ТУСУР, 2009. — 230 с. — Режим доступа: http://e.lanbook.com/book/4971 — Загл. с экрана.

18. Мартемьянов, Ю.Ф. Операционные системы. Концепции построения и обеспечения безопасности [Электронный ресурс] : учеб. пособие / Ю.Ф. Мартемьянов, А.В. Яковлев, А.В. Яковлев. — Электрон. дан. — Москва : Горячая линия-Телеком, 2011. — 332 с. — Режим доступа: https://e.lanbook.com/book/5176. — Загл. с экрана.

19. Кирнос, В.Н. Основы программирования на языке Ассемблера [Электронный ресурс] : учеб. пособие — Электрон. дан. — Москва : ТУСУР, 2007. — 106 с. — Режим доступа: https://e.lanbook.com/book/11624. — Загл. с экрана.

Учебное издание

А.С. Романов

Системное программирование

Методические указания по лабораторным работам, практическим занятиям, самостоятельной и индивидуальной работе
для студентов специальностей
10.03.01 – «Информационная безопасность», 10.05.02 – «Информационная безопасность телекоммуникационных систем», 10.05.04 – «Информационно-аналитические системы безопасности», 10.05.03 – «Информационная безопасность автоматизированных систем», 38.05.01 – «Экономическая безопасность»